TUGAS AKHIR – KI141502

DESAIN DAN ANALISIS ALGORITMA MULTIPLE STRING MATCHING PADA PENYELESAIAN PROBLEM KLASIK 12713

TRACY FILBERT RIDWAN
NRP 5111100176

Dosen Pembimbing I
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing II
Rully Soelaiman, S.Kom., M.Kom.

Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

UNDERGRADUATE THESIS – KI141502

DESIGN AND ANALYSIS MULTIPLE STRING MATCHING ALGORITHM FOR SOLVING SPOJ CLASSIC PROBLEM 12713

TRACY FILBERT RIDWAN
NRP 5111100176

First Advisor
Arya Yudhi Wijaya, S.Kom., M.Kom.

Second Advisor
Rully Soelaiman, S.Kom., M.Kom.

Department of Informatics
Faculty of Information Technology
Sepuluh Nopember Institute of Technology
Surabaya 2015

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS ALGORITMA MULTIPLE STRING MATCHING PADA PENYELESAIAN PROBLEM SPOJ KLASIK 12713

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh
TRACY FILBERT RIDWAN
NRP : 5111 100 176

Disetujui oleh Dosen Pembimbing Tugas Akhir:

1. Arya Yudhi Wijaya, S.Kom., M.Kom. .....
   NIP: 198409042010121002 (Pembimbing 1)
2. Rully Soelaiman, S.Kom., M.Kom. .....
   NIP: 197002131994021001 (Pembimbing 2)

SURABAYA

JUNI, 2015

DESAIN DAN ANALISIS ALGORITMA MULTIPLE STRING MATCHING PADA PENYELESAIAN PROBLEM SPOJ KLASIK 12713

Nama : **Tracy Filbert Ridwan**
NRP : **5111100176**
Jurusan : **Teknik Informatika – FTIf ITS**
Dosen Pembimbing I : **Arya Yudhi Wijaya, S.Kom., M.Kom.**
Dosen Pembimbing II : **Rully Soelaiman, S.Kom., M.Kom.**

Abstrak

String Matching adalah permasalahan klasik tentang pencocokan dua buah string dikatakan sama atau apakah sebuah string merupakan substring dari string lainnya. Permasalahan ini sering dijumpai dalam pencarian kata dalam kamus, pencarian kata dalam suatu dokumen, dan perhitungan DNA.

Dalam penelitian ini akan mengimplementasikan struktur data Suffix Array yang dilengkapi dengan informasi Longest Common Prefix untuk menyelesaikan permasalahan String Matching. Struktur data Suffix Array dipilih karena memiliki kompleksitas waktu pembuatan linear dan kompleksitas memori linear.

Pada permasalahan String matching terkadang mempunyai suatu syarat khusus dalam proses pencariannya sebagai contoh string tersebut harus identik dan string tersebut harus berulang sebanyak k kali, sehingga harus dilakukan perhitungan tambahan untuk memenuhi syarat khusus tersebut. Pada penelitian ini struktur data suffix array akan dipakai untuk menyelesaikan proses pencarian substring yang berulang tepat sebanyak k kali pada suatu string yang diberikan.

Kata kunci: String Matching, Suffix Array, Longest Common Prefix.

DESIGN AND ANALYSIS MULTIPLE STRING MATCHING ALGORITHM FOR SOLVING SPOJ CLASSIC PROBLEM 12713

Name	**: Tracy Filbert Ridwan**
NRP	**: 5111100176**
Department	**: Informatics Engineering – FTIf ITS**
Advisor I	**: Arya Yudhi Wijaya, S.Kom., M.Kom.**
Advisor II	**: Rully Soelaiman, S.Kom., M.Kom.**

Abstract

String Matching is a classic problem about finding if two string is identic or checking if a string is a substring from the another string. We can see this kind problem when we want to try finding a word from a dictionary, finding a word in a document, or calculation process in DNA.

In this final test will try to implement a data structure called suffix array with an extra information about it's longest common prefix for solving string matching problem. Suffix Array is chosen because it's have linear time complexity to build and linear memory complexity.

In string matching problem we often face another conditional to fulfil in the searching process for example the string must be identical, and the string must repeated exactly at k times, so another process must be calculated to fulfil that extra condition. In This final test Suffix Array used to solve searching process about finding substring that repeated exactly k times in some string given

Keywords : String Matching, Suffix Array, Longest Common Prefix.

KATA PENGANTAR

Puji Syukur saya panjatkan kepada Tuhan Yang Maha Esa yang telah memberikan rahmat dan karunia-Nya, sehingga penulis dapat menyelesaikan penelitian yang berjudul:

Desain dan Analisis Algoritma Multiple String Matching pada Penyelesaian Problem SPOJ Klasik 12713

Penulis menyadari bahwa penulis tidak mungkin dapat menyelesaikan penelitian ini tanpa bantuan dan dukungan dari banyak pihak, baik secara langsung maupun tidak. Untuk itu, penulis ingin mengucapkan terima kasih dan penghormatan sebesar-besarnya kepada:

1. Kedua orang tua penulis. Terima kasih untuk Mama yang telah melahirkan penulis ke dunia ini. Terima kasih untuk Papa yang selalu mengajari penulis untuk selalu menjadi pribadi yang lebih baik.
2. Bapak Rully Soelaiman S.Kom., M.Kom, selaku dosen pembimbing penulis. Terima kasih atas semua waktu diskusi, saran, kritik dan pengalaman yang diberikan, baik itu mengenai Kehidupan, *problem solving*, dan hal-hal lainnya.
3. Bapak Arya Yudhi Wijaya, S.Kom, M.Kom selaku dosen pembimbing penulis. Terima kasih atas bantuan, saran, dan masukan selama pengerjaan penelitian ini.
4. Bapak Auzi Asfarian selaku teman dosen IPB. Terima kasih telah meluangkan waktu untuk mengajari penulis tentang tata penulisan.
5. Teman-teman, karyawan, dan dosen Teknik Informatika ITS tidak bisa disebutkan satu-persatu. Terima kasih atas segala bantuannya.

Penulis telah berusaha menyelesaikan penelitian ini sebaik mungkin, tetapi penulis mohon maaf apabila terdapat kesalahan maupun kelalaian yang penulis lakukan

Surabaya, Juni 2015

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR KODE SUMBER

BAB I
PENDAHULUAN

Pada bab ini penulis menjelaskan tentang latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi, dan sistematik penulisan.

1.1 Latar Belakang

Multiple String Matching adalah permasalahan dalam membandingkan beberapa *string* untuk menemukan apakah terdapat dua buah *string* yang serupa. Permasalahan ini sangat sering dijumpai terutama dalam bidang keilmuan komputer sains seperti pencarian sebuah kata dalam kamus. Namun, tidak menutup kemungkinan permasalahan ini tidak dapat dipakai untuk memodelkan permasalahan dari bidang keilmuan lain. Sebagai contoh, penentuan kecocokan dua rantai DNA, dan penelitian tentang komponen pembentuk sebuah DNA/Protein dapat dimodelkan ke dalam permasalahan *multiple string matching*.

Dalam penyelesaian *multiple string matching*, sudah banyak algoritma ataupun struktur data yang ditemukan untuk memudahkan dan mengoptimalkan proses pemecahan permasalahan seperti KMP [1], Rabin Karp [2], Suffix Array [3]. Akan tetapi apakah algoritma dan struktur data tersebut dapat mendukung penyelesaian masalah lain yang dimodelkan ke permasalahan *multiple string matching*? Terkadang permasalahan lain yang ingin dimodelkan ke dalam permasalahan *multiple string matching* memiliki syarat khusus lain sebagai contoh, *string* yang ingin dicari harus terkandung secara keseluruhan dalam *string* lainnya, *string* yang ingin dicari harus terdapat paling tidak *k* kali dalam *string* lainnya, dll.

Pada penelitian ini akan diselesaikan sebuah permasalahan *multiple string matching* yang memiliki suatu syarat khusus yang didasari dari permasalahan klasik 12713 Strings pada situs SPOJ [4]. Pada permasalahan tersebut, akan dicari berapa banyak *substring* unik A yang terdapat pada *string* B dan berulang sebanyak tepat *k* kali. Contoh jika A = "miss" dan B = "mississippi" dan akan dicari *substring* yang berulang sebanyak tepat dua kali maka akan ada tiga substring yang memenuhi kriteria tersebut yaitu "is", "iss", "ss".

Secara sepintas solusi dari permasalahan ini dapat ditemukan apabila dengan mengetahui semua kemungkinan *substring* yang dapat dibentuk dari *string* A dan *string* B, namun pencarian semua *substring* yang terbentuk setidaknya memerlukan kompleksitas waktu sebesar $O(n^2)$. Maka dalam pengerjaan penelitian ini akan dicoba untuk mencari solusi lain yang memiliki kompleksitas waktu lebih kecil dari $O(n^2)$.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam penelitian ini ada sebagai berikut:

1. Bagaimana menemukan *substring* dari sebuah *string* yang berulang tepat sebanyak *k* kali?
2. Bagaimana menemukan *substring* yang terdapat pada dua buah *string* yang berbeda?
3. Bagaimana menyelesaikan permasalahan klasik 12713 Strings pada situs SPOJ ?

1.3 Batasan Masalah

Permasalahan yang dibahas pada penelitian ini memiliki beberapa batasan, yaitu sebagai berikut:

1. Implementasi dilakukan dengan bahasa pemrograman C++.

2. Batas maksimum *string* yang dimasukkan sebanyak 2.
3. Batas maksimum jumlah karakter dari setiap *string* sebanyak 8000 karakter yang terdiri huruf alfabet non-kapital '`a`' – '`z`'.
4. Nilai *k* yang diberikan tidak lebih besar dari panjang *string* yang diberikan.

1.4 Tujuan

Tujuan dari pengerjaan penelitian ini adalah mendesain dan menganalisis algoritma *multiple string matching* pada penyelesaian problem SPOJ klasik 12713.

1.5 Manfaat

Dalam penelitian ini menawarkan algoritma yang efisien untuk menyelesaikan problem *multiple string matching* yang diadaptasi dari permasalahan SPOJ klasik 12713.

1.6 Metodologi

Berikut metodologi yang digunakan dalam penelitian ini:

1. Penyusunan proposal penelitian
   Pada tahap ini dilakukan penyusunan proposal penelitian yang berisi definisi permasalahan *multiple string matching* beserta gambaran umum mengenai solusi untuk permasalahan *multiple string matching*.
2. Studi literatur
   Pada tahap ini dilakukan studi literatur mengenai penyelesaian permasalahan *multiple string matching* dengan menggunakan struktur data *Suffix Array* yang dilengkapi dengan informasi *Longest Common Prefix*. Literatur yang digunakan antara lain *paper*, buku referensi, dan artikel yang didapatkan dari internet.

3. Desain
   Pada tahap ini dilakukan desain algoritma serta struktur data untuk menyelesaikan permasalahan *multiple string matching* dengan menggunakan struktur data *Suffix Array* yang dilengkapi dengan informasi *Longest Common Prefix*.
4. Implementasi
   Pada tahap ini dilakukan implementasi solusi untuk permasalahan *multiple string matching* berdasarkan analisis dan desain yang telah dilakukan.
5. Uji coba dan evaluasi
   Pada tahap ini dilakukan uji coba untuk menguji kebenaran dan kinerja dari implementasi yang telah dilakukan. Pengujian kebenaran dilakukan dengan beberapa *input* (kasus uji), kemudian membandingkan *output* dari hasil pengujian dengan *output* dari implementasi yang lain. Pengujian kinerja dilakukan dengan membandingkan kompleksitas yang didapat dari hasil pengujian dengan kompleksitas yang didapat dari hasil analisis. Selain itu, evaluasi dilakukan untuk mencari bagian-bagian yang masih bisa dioptimasi dan memperbaiki kesalahan yang muncul.
6. Penyusunan buku
   Pada tahap ini dilakukan penyusunan buku yang berisi dokumentasi mengenai solusi untuk permasalahan *multiple string matching* dengan menggunakan struktur data *Suffix Array* yang dilengkapi dengan informasi *Longest Common Prefix*.

1.7 Sistematika Laporan

Berikut sistematika penulisan buku ini:

1. **BAB I : PENDAHULUAN**
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi, dan sistematika penulisan.
2. **BAB II : TINJAUAN PUSTAKA**
   Bab ini berisi dasar teori mengenai permasalahan dan algoritma yang digunakan dalam penelitian.
3. **BAB III : DESAIN**
   Bab ini berisi desain algoritma serta struktur data yang digunakan dalam penelitian.
4. **BABI IV : IMPLEMENTASI**
   Bab ini berisi implementasi berdasarkan desain algoritma serta struktur data yang telah dilakukan
5. **BAB V : UJI COBA DAN EVALUASI**
   Bab ini berisi uji coba dan evaluasi dari implementasi yang telah dilakukan.
6. **BAB VI : KESIMPULAN**
   Bab ini berisi kesimpulan dari hasil uji coba yang telah dilakukan.

(Halaman ini sengaja dikosongkan)

BAB II
TINJAUAN PUSTAKA

Pada bab ini akan dijelaskan dasar teori yang digunakan dalam menyelesaikan masalah yang ada di penelitian ini.

2.1 Definisi

Ditentukan *string S* adalah deret berhingga dari karakter-karakter yang termasuk dalam alphabet $\sum$, dapat ditulis $S \in \sum^*$. *S* diasumsikan selalu diakhiri oleh karakter sentinel \$, dimana \$ adalah karakter terkecil pada *S*.

Komparasi 2 *string A* dan *B* secara leksikografi dilakukan sebagai berikut. Apabila ada *j* yaitu indeks terkecil dimana $A[j] \neq B[j]$, perbandingan leksikografi antara *A* dan *B* sama dengan perbandingan leksikografi antara $A[j] = B[j]$.

Concatenation (penggabungan) dari 2 *string A* dan *B* adalah deret karakter-karakter *A* dilanjutkan dengan karakter-karakter *B*. Sebagai contoh untuk *A* = `hello$` dan *B* = `world$` maka *AB* = `helloworld$`. Dapat dilihat sentinel *A* tidak diikutkan. Penggabungan juga bisa dilakukan antara *string* dan karakter, sebagai contoh, untuk *c* = `s` maka *Bc* = `worlds$`.

String B adalah *substring* dari *S* apabila ada *string C* dan *D* dimana *S* = *CBD*.*C* dan *D* bisa merupakan *string* kosong (*empty string*).

Untuk *string S*, $S[i]$ adalah karakter ke-*i* pada *S*, dan $S[i...j] = S[i]S[i + 1]\ldots S[j]$. $0 \leq i \leq j < |S|$ dan $S[|S| - 1] =$ `$`. Sebagai contoh, untuk *string S* = `helloworld$`, $S[2]$ = `l` dan $S[1..5]$ = `ellow$`.

Suffix (akhiran) adalah *substring* yang diakhiri dengan sentinel. *suf*(*S*, *i*) adalah $S[i...|S| - 1]$. Sementara *prefix* (awalan) adalah *substring* yang dimulai dari indeks ke 0, *pre*(*S*, *i*) = $S[0\ldots i]$.

2.2 Suffix Array

Suffix Array (larik akhiran) adalah struktur data yang berisi *suffix-suffix* yang ter urut secara leksikografi dari sebuah *string S*. **Gambar 2.1** menunjukkan (a) contoh *suffix-suffix* dan (b) *Suffix Array* untuk *string S* = `mississippi$`.

a)

Suffix											
m	i	s	s	i	s	s	i	p	p	i	$
	i	s	s	i	s	s	i	p	p	i	$
		s	s	i	s	s	i	p	p	i	$
			s	i	s	s	i	p	p	i	$
				i	s	s	i	p	p	i	$
					s	s	i	p	p	i	$
						s	i	p	p	i	$
							i	p	p	i	$
								p	p	i	$
									p	i	$
										i	$
											$

b)

Suffix Array											
$											
i	$										
i	p	p	i	$							
i	s	s	i	p	p	i	$				
i	s	s	i	s	s	i	p	p	i	$	
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$									
p	p	i	$								
s	i	p	p	i	$						
s	i	s	s	i	p	p	i	$			
s	s	i	p	p	i	$					
s	s	i	s	s	i	p	p	i	$		

Gambar 2.1 Contoh *Suffix Array* untuk *String* `mississippi$`

Dalam implementasinya, *Suffix Array* tidak harus menyimpan *suffix-suffix* tersebut. *Suffix Array* direpresentasikan dalam *array integer SA*, dimana $SA[i] = x$ apabila $suf(S, x)$ adalah *suffix* ke – *i* pada *Suffix Array*.

Selain *SA*, *Suffix Array* juga dilengkapi dengan larik *ISA (invers Suffix Array)* dimana $ISA[SA[i]] = i$ (**Gambar 2.2**). Pada pembahasan penelitian ini selanjutnya, algoritma dan pendekatan *Suffix Array* yang digunakan adalah *Suffix Array – Induced Sorting* yang dipublikasikan pada jurnal referensi [3] dan dibahas lebih detail pada buku Tugas Akhir [5].

i	SA[i]	ISA[i]	*Suf(S,SA[i])*										
0	11	5	$										
1	10	4	i	$									
2	7	11	i	p	p	i	$						
3	4	9	i	s	s	i	p	p	i	$			
4	1	3	i	s	s	i	s	s	i	p	p	i	$
5	0	10	m	i	s	s	i	s	s	i	p	p	i	$
6	9	8	p	i	$								
7	8	2	p	p	i	$							
8	6	7	s	i	p	p	i	$					
9	3	6	s	i	s	s	i	p	p	i	$		
10	5	1	s	s	i	p	p	i	$				
11	2	0	s	s	i	s	s	i	p	p	i	$	

Gambar 2.2 Contoh Larik *SA*, *ISA* pada *Suffix Array* untuk *String* `mississippi$`

2.3 Relasi Suffix Array dengan Longest Common Prefix

Longest Common Prefix (LCP) adalah jumlah karakter *prefix* maksimum yang serupa dari dua buah *string*. Sebagai contoh *string A* = `mississippi$` dan *string B* = `miss$` maka *lcp*(*A, B*) = 3 (**Gambar 2.3**). Sering kali informasi LCP ditambahkan ke dalam *Suffix Array* untuk menyelesaikan suatu permasalahan tertentu seperti yang akan dijelaskan pada Subbab 2.6 *k* – Occurrence dan 2.7 Common Occurrence.

m	i	s	s	i	s	s	i	p	p	i	$
m	i	s	t	y	c	$					

Gambar 2.3 Contoh LCP untuk *String* `mississippi$` dan `mistyc$`

Dalam *Suffix Array* informasi LCP di representasikan dalam sebuah *array integer*, dimana $LCP[i] = lcp(SA[i], SA[i\text{-}1])$ dan $LCP[0] = 0$ (**Gambar 2.4**). Untuk mengetahui LCP dari dua buah *suffix* pada *Suffix Array* dapat diketahui dengan $lcp(SA[x], SA[z]) = \min_{x<y\leq z}\{lcp[y]\}$ dimana x dan z adalah suatu indeks pada *Suffix Array* $0 \leq x, z \leq |S|$.

i	SA[i]	ISA[i]	*LCP*[i]	*Suf(S,SA[i])*										
0	11	5	0	$										
1	10	4	0	i	$									
2	7	11	1	i	p	p	i	$						
3	4	9	1	i	s	s	i	p	p	i	$			
4	1	3	4	i	s	s	i	s	s	i	p	p	i	$
5	0	10	0	m	i	s	s	i	s	s	i	p	p	i	$
6	9	8	0	p	i	$								
7	8	2	1	p	p	i	$							
8	6	7	0	s	i	p	p	i	$					
9	3	6	2	s	i	s	s	i	p	p	i	$		
10	5	1	1	s	s	i	p	p	i	$				
11	2	0	3	s	s	i	s	s	i	p	p	i	$	

Gambar 2.4 Contoh Larik *SA*, *ISA*, dan *LCP* pada *Suffix Array* untuk *String* `mississippi$`

Saat nilai $nPref > 0$ dimana $nPref = lcp(SA[x], SA[z])$ berarti *suffix-suffix* pada indeks x hingga z di suatu *Suffix Array* memiliki karakter awal yang sama dengan panjang *nPref*. Pada **Gambar 2.5** sebagai contoh apabila $nPref = lcp(SA[2], SA[4]) = 3$ pada *Suffix Array* yang dibangun dari *string* `abcdbcdefbcdbd$`, maka *suffix-suffix* pada *Suffix Array* di indeks 2 hingga 4 memiliki 3 karakter pertama yang sama (lihat kolom-kolom yang diwarnai coklat). Jika nilai $nPref > 0$ maka *npref* karakter pertama pada $suf(S, SA[x])$ setidaknya berulang sebanyak m kali dimana $m = z - x + 1$. Untuk contoh lain $nPref =$

lcp(*SA*[9],*SA*[11]) = 1, karakter pertama pada *suf*(*S*,*SA*[9]) berulang setidaknya sebanyak 3 kali ($z - x + 1 = 11 - 9 + 1 = 3$) bisa dilihat pada **Gambar 2.5** pada kolom-kolom yang diwarnai merah.

i	*SA*[i]	*LCP*[i]	Suf(*S*, *SA*[i])													
0	14	0	$													
1	0	0	a	b	c	d	b	c	d	e	f	b	c	d	b	d	$
2	1	0	b	c	d	b	c	d	e	f	b	c	d	b	d	$	
3	9	4	b	c	d	b	d	$									
4	4	3	b	c	e	f	b	c	d	b	d	$					
5	12	1	b	d	$												
6	2	0	c	d	b	c	d	e	f	b	c	d	b	d	$		
7	10	3	c	d	b	d	$										
8	5	2	c	d	e	f	b	c	d	b	d	$					
9	13	0	d	$													
10	3	1	d	b	c	d	e	f	b	c	d	b	d	$			
11	11	2	d	b	d	$											
12	6	1	d	e	f	b	c	d	b	d	$						
13	7	0	e	f	b	c	d	b	d	$							
14	8	0	f	b	c	d	b	d	$								

Gambar 2.5 Contoh larik *SA* dan larik *LCP* pada *String* `abcdbcdefbcdbd$`

2.4 Fixed Size Range Minimum Queries

Range Minimum Queries adalah sebuah permasalahan tentang pencarian data minimum di sebuah larik *integer A* dari rentang yang ditentukan. Permasalahan ini akan dihadapi apabila ada suatu kondisi dimana informasi LCP dua *suffix* pada S*uffix Array* ingin diketahui. Dalam pengerjaan penelitian ini ditemui permasalahan *Fixed Size Range Minimum Queries* untuk mendapat informasi *lcp*(*SA*[*x*], *SA*[*x*+*k*-1]) dimana $0 \leq x < |S|$, $0 < k \leq |S|$, $x+k-1 \leq |S|$ dan *k* tidak pernah berubah, dengan kompleksitas di bawah

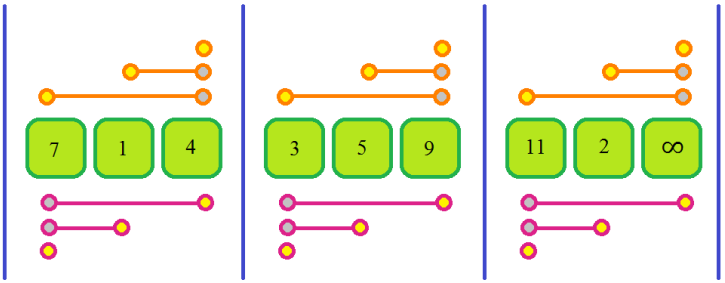
O(k). Sehingga metode *Fixed Segment Array* digunakan untuk mencari nilai $lcp(SA[x], SA[x+k-1]) = \min_{x<y\leq x+k-1}\{LCP[y]\}$.

Dalam metode ini setiap elemen pada larik A akan dibagi ke dalam beberapa segmen dengan besar m dimana m adalah besar jangkauan yang ingin diketahui untuk setiap *query* yang diberikan.

Apabila jumlah elemen pada larik A tidak pas dibagi ke dalam segmen m, tambahkan elemen baru pada akhir larik A dengan nilai ∞ sehingga jumlah elemen pada larik A dapat dibagi ke dalam segmen m.

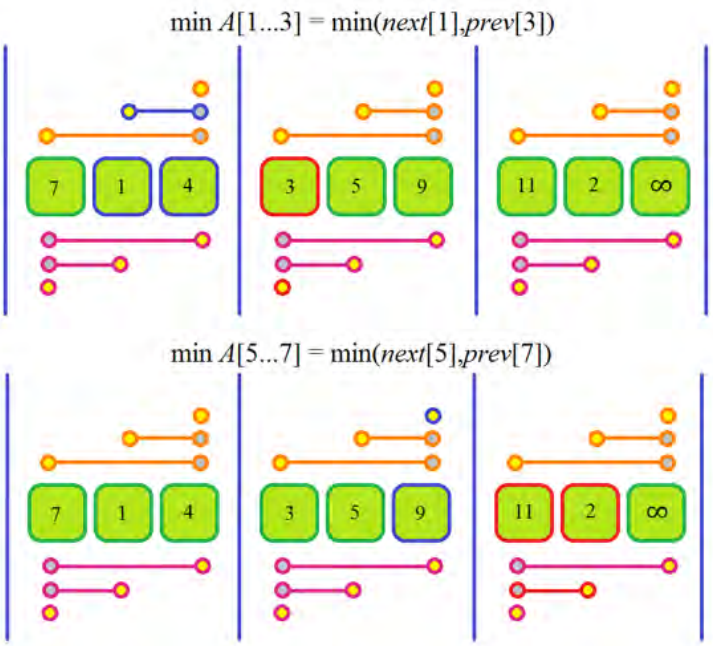
Setelah itu akan dibentuk larik *next* dan *prev* dimana larik *next*[i] menyimpan data minimum pada indeks i hingga batas akhir pada segmen tersebut dan *prev*[i] menyimpan data minimum pada indeks i hingga batas awal pada segmen tersebut.

Pada **Gambar 2.6** bisa dilihat garis jingga mewakili jangkauan yang dicakup oleh *next*[i] dan garis ungu mewakili jangkauan yang dicakup oleh *prev*[i] (i dilambangkan dengan titik kuning) pada larik A yang dibagi ke dalam segmen dengan besar m = 3.



Gambar 2.6 Ilustrasi Larik *next* dan Larik *prev* pada Suatu Larik *Integer* A dengan Jumlah Elemen 8 yang Dibagi ke Dalam Segmen dengan Besar m = 3

Dengan terbentuknya larik *next* dan *prev*. Informasi nilai minimum $A[i...i+k]$ bisa didapatkan dari nilai minimum($next[i]$, $prev[i+k-1]$). **Gambar 2.7** mengilustrasikan bagaimana nilai minimum dari larik $next[i]$ dan $prev[i+k-1]$ dapat merepresentasikan nilai minimum dari larik $A[i...i+k-1]$ dimana $m = 3$ dan larik $A = A[0], A[1],\ldots, A[7]$.



Gambar 2.7 Ilustrasi Pencarian Nilai Minimum A[1…3] dan A[5…7] dengan Menggunakan Larik *next* dan *prev*

Larik *next* dapat dibentuk dengan menelusuri larik *A* dari kiri ke kanan dan untuk larik *prev* dapat dibentuk dengan menelusuri larik *A* dari kanan ke kiri. Sehingga dibutuhkan operasi pra proses sebesar $2n$ dan untuk mencari informasi nilai minimum $A[i...i+m-1]$ dibutuhkan satu operasi yaitu membandingkan nilai

minimum dari *next*[*i*] dan *prev*[*i*+*m*-1] dimana *N* adalah jumlah elemen pada larik yang ingin dibentuk struktur *Fixed Segment Array*.

2.5 Relasi Fixed Segment Array dengan LCP

Fixed Segment Array adalah suatu struktur data yang digunakan untuk mengetahui nilai minimal dari suatu jangkauan *k* pada larik *integer* dengan kompleksitas waktu O(1) seperti yang telah dijelaskan pada subbab 2.4. Karena representasi larik LCP menggunakan larik *integer* maka struktur *Fixed Segment Array* dapat dimanfaatkan untuk kebutuhan pencarian data minimal pada suatu jangkauan *k*. Sebagai contoh nilai LCP dari (*SA*[*i*], *SA*[*i*+1], …, *SA*[*i*+*k*]) diketahui dengan mendapatkan min(*LCP*[*i*], *LCP*[*i*+1], …, *LCP*[*i*+*k*-1]), namun nilai LCP tersebut dapat diketahui dengan mencari nilai min(*next*[*i*], *prev*[*i*+*k*-1]) dengan bantuan *Fixed Segment Array*. Sehingga proses pencarian nilai LCP pada suatu jangkauan *k* dapat diketahui dengan membandingkan nilai larik *next* dan *prev,* bukan dari nilai larik *LCP*. Sehingga, mengakibatkan pencarian nilai *lcp*(*SA*[*i*], *SA*[*i*+1], …, *SA*[*i*+*k*]) hanya membutuhkan kompleksitas waktu sebesar O(1) dan bukan O(*k*).

i	SA[i]	ISA[i]	LCP[i]	*next*[i]	*prev*[i]	*Suf(S,SA[i])*									
0	11	5	0	0	0	$									
1	10	4	0	0	0	i	$								
2	7	11	1	1	0	i	p	p	i	$					
3	4	9	1	0	1	i	s	s	i	p	p	i	$		
4	1	3	4	0	1	i	s	s	i	s	s	i	p	p	i	$	
5	0	10	0	0	0	m	i	s	s	i	s	s	i	p	p	i	$
6	9	8	0	0	0	p	i	$									
7	8	2	1	0	0	p	p	i	$								
8	6	7	0	0	0	s	i	p	p	i	$						
9	3	6	2	1	2	s	i	s	s	i	p	p	i	$			
10	5	1	1	1	1	s	s	i	p	p	i	$					
11	2	0	3	3	1	s	s	i	s	s	i	p	p	i	$		

Gambar 2.8 Contoh Larik *next* dan *prev* pada Larik *LCP* dari *String* `mississippi$` dengan $k = 4$

Pada **Gambar 2.8** dapat dilihat representasi larik *next* dan *prev* pada *LCP* yang dibentuk dari string `mississippi$`. Karena nilai k = 4 maka besar blok yang akan dibentuk pada struktur *fixed segment array* adalah m = 3, ini dikarenakan nilai $lcp(SA[i]\ldots SA[i+k-1])$ = $\min(LCP[i+1]\ldots LCP[i+k-1])$ dan dalam jangkauan i+1 hingga i+k-1 terdapat k-1 elemen yang harus diperiksa nilainya yang mengakibatkan panjang segmen yang dibentuk sebesar k-1 dan bukan k.

2.6 *k* – Occurrence

k-Occurrence adalah informasi yang bisa didapat dari LCP pada *Suffix Array*. Dimana tujuan *k-Occurrence* untuk mengetahui *substring* mana saja yang berulang sebanyak tepat k kali dari sebuah *string S*. Informasi *k -Occurrence* dapat direpresentasikan dengan sebuah larik *integer* 2 dimensi *KO*, dimana $KO[i][0]$ dan $KO[i][1]$ masing-masing merepresentasikan *substring – substring* pada batas $S[i\ldots i+KO[i][0]-1]$ hingga $S[i\ldots i+KO[i][1]-1]$ berulang tepat sebanyak k kali dan $KO[i][0] = KO[i][1] = -1$ apabila tidak ada *substring* yang berulang tepat sebanyak k kali

pada *suf*(*S*, *i*). **Gambar 2.9** dan **Gambar 2.10** mengilustrasikan larik *KO* pada *string* = `misssissippi$` dengan nilai *k* = 2.

i	KO[*i*][0]	KO[*i*][0]	*Suf(S,i)*											
0	-1	-1	m	i	s	s	i	s	s	i	p	p	i	$
1	-1	-1	i	s	s	i	s	s	i	p	p	i	$	
2	-1	-1	s	s	i	s	s	i	p	p	i	$		
3	-1	-1	s	i	s	s	i	p	p	i	$			
4	2	4	i	s	s	i	p	p	i	$				
5	2	3	s	s	i	p	p	i	$					
6	2	2	s	i	p	p	i	$						
7	-1	-1	i	p	p	i	$							
8	1	1	p	p	i	$								
9	-1	-1	p	i	$									
10	-1	-1	i	$										
11	-1	-1	$											

Gambar 2.9 Ilustrasi Larik *KO* pada *String* `mississippi$`

i	SA[i]	*KO*[SA[i]][0]	*KO*[SA[i]][1]	*Suf(S,SA[i])*										
0	11	-1	-1	$										
1	10	-1	-1	i	$									
2	7	-1	-1	i	p	p	i	$						
3	4	2	4	i	s	s	i	p	p	i	$			
4	1	-1	-1	i	s	s	i	s	s	i	p	p	i	$
5	0	-1	-1	m	i	s	s	i	s	s	i	p	p	i	$
6	9	-1	-1	p	i	$								
7	8	1	1	p	p	i	$							
8	6	2	2	s	i	p	p	i	$					
9	3	-1	-1	s	i	s	s	i	p	p	i	$		
10	5	2	3	s	s	i	p	p	i	$				
11	2	-1	-1	s	s	i	s	s	i	p	p	i	$	

Gambar 2.10 Ilustrasi Larik *KO* pada *String* `mississippi$` pada *Suffix Array*

Dapat dilihat bahwa saat nilai *kPrefRight* > *0* dimana *kPrefRight* = *lcp*(*SA*[*i*]… *SA*[*i*+*k*-1]) dan *i* adalah suatu indeks pada S*uffix Array*, semua *prefix* pada *substring S*[*SA*[*i*]… *SA*[*i*] + *kPrefRight* - 1] berulang paling tidak sebanyak *k* kali. Jika

informasi yang ingin didapat berupa *substring y*ang berulang sebanyak tepat *k* kali maka saat nilai *kPrefLeft* ≤ *kPrefRight* dimana *kPrefLeft* = *max*(*lcp(SA*[*i*-1], *SA*[*i*])+1, *lcp*(*SA*[*i*+*k-1*], *SA*[*i*+*k*])) berarti semua *prefix*(*TS,x-1*) dimana *TS* = *substring S*[*SA*[*i*]…*SA*[*i*] + *kPrefRight* -1] dan *kPrefLeft* ≤ *x* ≤ *kPrefRight* berulang tepat sebanyak *k* kali. Dengan demikian, kompleksitas waktu yang dibutuhkan untuk pembentukan larik *KO* sebesar O(|*S*|), kompleksitas waktu pembentukan larik *SA string S* sebesar O(|*S*|), kompleksitas waktu pembentukan larik *LCP string S* sebesar O(|*S*|), dan kompleksitas waktu pembentukan *Fixed Segment Array* sebesar O(2|*S*|). Oleh karena itu total kompleksitas waktu yang diperlukan untuk pembentukan larik *KO* sebesar O(|*S*|).

2.7 Common Occurrence

Common Occurrence adalah informasi yang berguna untuk menemukan semua posisi *substring A* yang ada pada *string B*. Informasi ini akan ditampung pada sebuah larik *integer CO*[*i*] dimana 0 ≤ *i* < |*B*|. Saat nilai *CO*[*i*] > 0, semua *prefix pre*(*suf*(*B, i*), *x-1*) merupakan *substring* dari *string A* dimana 0 < *x* ≤ *CO*[*i*]. **Gambar 2.11** mengilustrasikan larik *CO* saat *string A* = `mistype$` dan *string B* = `mississippi$`.

i	*CO[i]*	*Suf(S,i)*											
0	3	m	i	s	s	i	s	s	i	p	p	i	$
1	2	i	s	s	i	s	s	i	p	p	i	$	
2	1	s	s	i	s	s	i	p	p	i	$		
3	1	s	i	s	s	i	p	p	i	$			
4	2	i	s	s	i	p	p	i	$				
5	1	s	s	i	p	p	i	$					
6	2	s	i	p	p	i	$						
7	0	i	p	p	i	$							
8	1	p	p	i	$								
9	1	p	i	$									
10	1	i	$										
11	0	$											

Gambar 2.11 Contoh Larik CO saat *String* A = `mistype$` dan *String* B = `mississippi$`

Common Occurrence dapat dibangun dengan menggunakan informasi LCP pada *Suffix Array* dari *string C* dimana *string C* adalah hasil *concatenation* dari *string A,* sebuah karakter sentinel lain selain `$`, dan *string B*. **Gambar 2.12** mengilustrasikan *Suffix Array* dan LCP dari *string C* dimana *string A* = `mistype$`, *B* = `mississippi$` dan karakter `%` merupakan karakter sentinel yang disisipi di antara *string A* dan *B*, baris yang berwarna biru menandakan *suffix* dari *string A* dan baris yang berwarna kuning menandakan *suffix* dari *string B*.

Nilai indeks *i* pada *CO*[*i*] sebanding dengan nilai *SA*[*x*]-|*A*|-1 dimana *x* merupakan indeks *Suffix Array string C* dan $0 \leq x <$ |*C*|, untuk setiap *suffix* yang merepresentasikan *suffix* dari *string B* atau nilai *SA*[*x*] > |*A*|. Nilai *CO*[*i*] dapat dihitung dengan mencari nilai maksimum dari *lcp(SA*[*left*]…*SA*[*x*]) dan *lcp(SA*[*x*]…*SA*[*right*]). Dimana *left* dan *right* merupakan indeks terdekat dengan *x* untuk *SA*[*left*], *SA*[*right*] <|*A*|, *left* < *x* < *right.* Apabila tidak ada nilai *left, right* yang memenuhi maka nilai *lcp(SA*[*left*]…*SA*[*x*]) = 0 atau *lcp(SA*[*x*]…*SA*[*right*]) = 0.

i	SA[*i*]	*LCP*[*i*]	*suf*(*C*,*i*)																			
0	19	0	$																			
1	7	0	%	m	i	s	s	i	s	s	i	p	p	i	$							
2	6	0	e	%	m	i	s	s	i	s	s	i	p	p	i	$						
3	18	0	i	$																		
4	15	1	i	p	p	i	$															
5	12	1	i	s	s	i	p	p	i	$												
6	9	4	i	s	s	i	s	s	i	p	p	i	$									
7	1	2	i	s	t	y	p	e	%	m	i	s	s	i	s	s	i	p	p	i	$	
8	8	0	m	i	s	s	i	s	s	i	p	p	i	$								
9	0	3	m	i	s	t	y	p	e	%	m	i	s	s	i	s	s	i	p	p	i	$
10	5	0	p	e	%	m	i	s	s	i	s	s	i	p	p	i	$					
11	17	1	p	i	$																	
12	16	1	p	p	i	$																
13	14	0	s	i	p	p	i	$														
14	11	2	s	i	s	s	i	p	p	i	$											
15	13	1	s	s	i	p	p	i	$													
16	10	3	s	s	i	s	s	i	p	p	i	$										
17	2	1	s	t	y	p	e	%	m	i	s	s	i	s	s	i	p	p	i	$		
18	3	0	t	y	p	e	%	m	i	s	s	i	s	s	i	p	p	i	$			
19	4	0	y	p	e	%	m	i	s	s	i	s	s	i	p	p	i	$				

Gambar 2.12 Contoh *Suffix Array* dan LCP pada *String* C.

Nilai *CO*[*i*] dapat dicari dengan menyisiri larik *LCP* dari kiri ke kanan dan dilanjutkan menyisi larik *LCP* dari kanan ke kiri sehingga kompleksitas waktu yang dibutuhkan sebesar O(2|*B*|) untuk pembentukan larik *CO*, kompleksitas waktu pembentukan larik *SA string C* sebesar O(|*C*|), dan kompleksitas waktu pembentukan larik *LCP string C* sebesar O(|*C*|). Sehingga total kompleksitas waktu yang diperlukan untuk pembentukan larik *KO* sebesar O(|*C*|) atau O(|*A*|+|*B*|).

2.8 Sphere Online Judge (SPOJ)

Sphere Online Judge (SPOJ) adalah sistem penilaian *online* yang berisi berbagai jenis permasalahan pemrograman yang dapat diselesaikan oleh pengguna dengan mengirim kode sumber program yang berisi solusi dari permasalahan yang diberikan. Setiap permasalahan mempunyai format masukan yang diberikan dan format keluaran yang diminta. Selain itu setiap permasalahan

juga mempunyai batasan tertentu termasuk lingkungan penilaian, batasan waktu, batasan memori, dan batasan dari permasalahan yang dideskripsikan.

Kode sumber program yang diterima sistem akan dikompilasi dan dijalankan pada lingkungan penilaian sistem. Program akan diuji menggunakan masukan dari berkas masukan permasalahan kemudian hasil keluaran program akan dibandingkan dengan berkas keluaran permasalahan. Sistem akan memberikan umpan balik kepada pengguna antara lain:

1. ***Accepted***, artinya program tidak melanggar batasan yang diberikan dan hasil keluaran program sama dengan berkas keluaran permasalahan.
2. ***Wrong Answer***, artinya hasil keluaran program tidak sama dengan berkas keluaran permasalahan.
3. ***Time Limit Exceeded***, artinya program melanggar batasan waktu yang diberikan.
4. ***Memory Limit Exceeded***, artinya program melanggar batasan memori yang diberikan.
5. ***Runtime Error***, artinya terjadi *error* pada saat program dijalankan.
6. ***Compile Error***, artinya terjadi *error* pada saat kompilasi kode sumber program.

Selain itu sistem juga memberikan umpan balik kepada pengguna mengenai waktu dan memori yang dibutuhkan program pada saat diuji menggunakan masukan dari berkas masukan permasalahan.

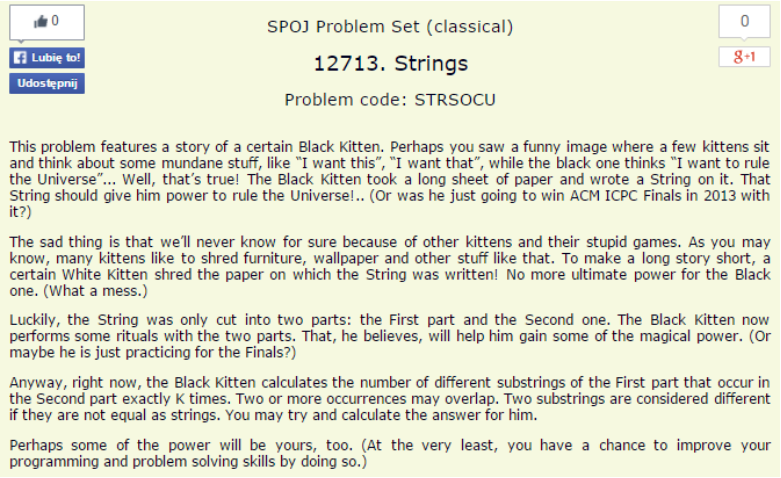
2.9 Penjelasan Permasalahan dan Strategi Penyelesaian

Permasalahan yang ingin diselesaikan dalam penelitian ini adalah permasalahan SPOJ klasik 12713 *Strings*. Detail permasalahan dan strategi penyelesaian akan dijelaskan pada

Subbab 2.9.1 Permasalahan SPOJ Klasik 12713 *Strings* dan 2.9.2 Strategi Penyelesaian Permasalahan SPOJ Klasik 12713 *Strings*.

2.9.1 Permasalahan SPOJ Klasik 12713 *Strings*

Salah satu permasalahan yang terdapat pada SPOJ adalah *Strings* yang mempunyai nomor 12713 dan kode STRSOCU. Deskripsi permasalahan ditunjukkan dalam **Gambar 2.13**.



Gambar 2.13 Deskripsi Permasalahan *Strings*

Deskripsi singkat permasalahan tersebut adalah diberikan dua *input string A, B* dan sebuah *integer k.* Cari berapa banyak *substring* unik dari *string A* yang ada pada *string B* yang berulang sebanyak tepat *k* kali pada *string B*. Berikut merupakan format masukan dari permasalahan tersebut:

1. Masukan terdiri atas beberapa kasus uji, format masukan setiap kasus uji didefinisikan pada poin 2 hingga poin 4.
2. Baris pertama berisi sebuah *string A* yang hanya terdiri atas huruf alfabet kecil '`a`' – '`z`'.

3. Baris kedua berisi sebuah *string B* yang hanya terdiri atas huruf alfabet kecil ‘a’ – ‘z’.
4. Baris ketiga berisi sebuah bilangan *integer k* yang merepresentasikan jumlah perulangan *substring* yang ingin dicari.

Berikut merupakan format keluaran dari permasalahan tersebut:

1. Untuk setiap kasus uji keluarkan sebuah *integer W* yang merepresentasikan jumlah *substring* unik dari *string A* yang muncul pada *string B* dan berulang tepat sebanyak *k* kali pada *string B*.

Berikut merupakan batasan dari permasalahan tersebut:

1. $|A|,|B| \leq 8.000$
2. $k \leq |B|$

2.9.2 Strategi Penyelesaian Permasalahan SPOJ Klasik 12713 *Strings*

Permasalahan ini dapat diselesaikan menggunakan informasi *k-Occurrence* dan *Common Occurrence* yang sudah dijelaskan pada Subbab 2.6 *k* – Occurrence dan 2.7 Common Occurrence. Dari informasi *k-Occurrence* dan *Common Occurrence* dapat diketahui *substring B* yang berulang tepat sebanyak *k* kali dan *substring A* yang muncul pada *string B*. Saat nilai $KO[i][0] \leq CO[i]$ berarti terdapat *n* karakter pertama yang memenuhi syarat permasalahan pada *suf(B*, *i*) dimana $n = \min(CO[i],KO[i][1]) - KO[i][0] + 1$, $KO[i][0] \neq -1$. Jumlah *substring* unik yang sesuai dengan deskripsi permasalahan dapat dengan menjumlahkan *n* untuk setiap $0 \leq i < |B|$. Dengan strategi ini dapat dilihat kompleksitas waktu untuk solusi permasalahan SPOJ klasik 12713 *Strings* ialah sebesar $O(|B|)$. Akan tetapi, karena dalam perhitungannya diperlukan informasi *k-Occurrence* dan *Common Occurrence* yang masing-masing memiliki kompleksitas waktu $O(2|B|)$ dan $O(|A| + |B|)$. Oleh karena itu, dapat

dikatakan bahwa kompleksitas waktu untuk keseluruhan proses perhitungan sebesar O(n) dimana n merupakan jumlah karakter pada *string* A dan B. Sehingga algoritma tersebut dapat diterima karena, batas waktu yang diberikan *problem* adalah sebesar 2 sekon dan kompleksitas solusi yang diberikan terkait batasan pada permasalahan adalah $|A| = 8.000$, $|B| = 8.000$.

(Halaman ini sengaja dikosongkan)

BAB III
DESAIN

Pada bab ini akan dijelaskan mengenai metodologi yang digunakan dalam implementasi algoritma pada penelitian ini.

3.1 Deskripsi Umum Program

Program akan menerima masukan berupa banyak data uji, dan untuk setiap masukan data uji terdiri atas *string A, string B,* sebuah bilangan bulat *integer* positif *k*. Kemudian sistem akan melakukan proses pembentukan larik *k-Occurrence* dan larik *Common Occurrence* dari hasil konstruksi *LCP* dari masing-masing *string* masukan untuk melakukan kalkulasi jumlah *substring A* yang berulang sebanyak *k* kali pada *substring B*. *Pseudocode* Fungsi Main ditunjukkan dalam **Gambar 4.1**.

```
Main()
1.  T ← Jumlah data uji
2.  For  t ← 0 to T-1
3.      A ← Masukan string A
4.      B ← Masukan string B
5.      K ← Masukan K
6.      Pembuatan SA & LCP dari string B
7.      Pembuatan Larik k - Occurrence
8.      Pembuatan SA & LCP dari string hasil concatenation A & B
9.      Pembuatan Larik Common Occurrence
10.         Proses kalkulasi jawaban
```

Gambar 4.1 *Pseudocode* Fungsi Main

3.2 Desain algoritma

Program terdiri atas 5 fungsi utama Konstruksi *Longest Common Prefix, Fixed Segment Array, k-Occurrence, Common Occurrence,* dan proses kalkulasi jawaban. Pada Subbab ini akan dijelaskan desain algoritma dari masing-masing fungsi utama tersebut.

3.2.1 Desain Fungsi Konstruksi *Longest Common Prefix*

LCP adalah informasi tambahan yang nantinya akan dicari dalam suatu *Suffix Array,* yang nantinya informasi tersebut digunakan konstruksi larik *k-Occurrence* & *Common Occurrence* seperti yang telah dijelaskan pada Subbab 2.6 *k* – Occurrence dan 2.7 Common Occurrence.

```
/***
* S = input string
* SA = Suffix Array  dari string S
* ISA = Inverse Suffix Array dari string S
* n = |S|
* LCP = larik LCP untuk string S
***/
ComputeLCPArray(S, SA, ISA, n, LCP)
```

```
1.  h = 0
2.  for i ← 0 to n -1
3.      pos = ISA[i]
4.      j = SA[pos-1]
5.      hbound = min(n-j, n-i)
6.      while( h<hbound && S[i+h] == S[j+h]) h++;
7.      LCP[pos] = h
8.      if(h > 0) h--
9.  LCP[0] = 0
```

Gambar 4.2 *Pseudocode* Fungsi Konstruksi LCP

Fungsi *ComputeLCPArray* digunakan untuk mencari nilai LCP dari setiap *suffix* yang bersebelahan pada *Suffix Array* seperti yang telah dijelaskan pada Subbab 2.3 Relasi *Suffix Array* dengan *Longest Common Prefix*. *Pseudocode* fungsi pembuatan larik *LCP* ditunjukkan dalam **Gambar 4.2.**

3.2.2 Desain Fungsi Konstruksi Struktur *Fixed Segment Array*

Fixed Segment Array adalah sebuah struktur data yang dipakai dalam konstruksi larik *k-Occurrence*. Struktur data ini dibentuk dengan tujuan mengoptimalkan proses konstruksi larik *k-Occurrence,* dimana saat konstruksi larik *k-Occurrence* diperlukan nilai *lcp*(*SA*[*i*]… *SA*[*i*+*k*-1]) dan proses pencarian nilai LCP tersebut sama dengan permasalahan yang telah dibahas pada Subbab 2.4. *Pseudocode* fungsi pembuatan *Fixed Segment array* ditunjukkan dalam **Gambar 4.3**.

Informasi yang ingin diketahui adalah informasi $lcp(SA[x], SA[x+k-1]) = \min_{x<y\leq x+k-1}\{LCP[y]\}$, sehingga bisa dilihat jangkauan yang ingin diketahui adalah *k*-1 (baris 1). Apabila besar larik *LCP* tidak bisa dibagi ke dalam blok-blok dengan panjang *k*-1, maka pada akhir larik *LCP* akan ditambahkan dengan nilai 0 (dalam Subbab 2.4 dijelaskan bahwa penambahan nilai diisi dengan nilai tak hingga, namun pada kasus ini nilai *i*+*k*-1 dipastikan tidak melebihi panjang *string* yang diberikan dan untuk menjaga konsistensi nilai LCP maka nilai *LCP*[*i*] = 0 untuk *i* > *n*) hingga besar larik *LCP* dapat dibagi ke dalam blok dengan panjang *k*-1(baris 1 hingga baris 6). Untuk konstruksi larik *prev* dapat dibentuk dengan cara menelusuri larik *LCP* dari depan ke belakang dan sebaliknya untuk membentuk larik *next*(baris 7 sampai baris 16) seperti yang dijelaskan pada Subbab 2.4.

```
/***
* S = string input B
* k = jumlah occurrence yang diinginkan
* n = |S|
* LCP = larik LCP untuk Suffix Array  S
* prev = larik prev untuk segment array
* next = larik next untuk segment array
***/
BuildFixedSegmentArray(k, n, LCP, prev, next)
```

```
1.  rep = k – 1
2.  temp = n/rep
3.  if(n%rep) temp++
4.  arraySize = temp * rep
5.  for i ← n to arraySize
6.      LCP[i] = 0
7.  for i ← 0 to arraySize
8.      if(i%rep == 0)
9.          prev[i] = LCP[i]
10.     else
11.         prev[i] = min(prev[i-1],LCP[i])
12. for i ← arraySize to 0
13.     if(i%rep == rep-1)
14.         next[i] = LCP[i]
15.     else
16.         next[i] = min(next[i+1],LCP[i])
```

Gambar 4.3 *Pseudocode* Fungsi Konstruksi Fixed Segment Array

3.2.3 Desain Konstruksi Larik *k-Occurrence*

Larik *k-Occurrence* adalah sebuah informasi yang bisa didapat dengan hasil perhitungan menggunakan informasi LCP pada suatu *Suffix Array* seperti yang telah dijelaskan pada Subbab 2.6. *Pseudocode* Konstruksi larik *k-Occurrence* ditunjukkan dalam **Gambar 4.4**.

Pada larik *k-Occurrence* nilai *KO*[*SA*[*i*]][0] dan *KO*[*SA*[*i*]][1] masing-masing merepresentasikan nilai *kPrefLeft* dan *kPrefRight*. Seperti yang telah dijelaskan pada Subbab 2.6 informasi *kPrefRight* didapat dari $\min_{i<y\leq i+k-1}\{LCP[y]\}$. Apabila nilai *kPrefRight* > 0 berarti setidaknya terdapat *kPrefRight* karakter pertama pada *suf*(*S*, *i*) yang berulang setidaknya *k* kali. Untuk *k* = 1 tidak diperlukan informasi *kPrefRight* dari $\min_{i<y\leq i+k-1}\{LCP[y]\}$ karena setidaknya *suf*(*S*, *i*) berulang 1 kali hingga nilai *kPrefRight* untuk *k = 1* dapat diketahui dengan panjang dari *suf*(*S*, *i*). Dapat dilihat pada baris 1 struktur *Fixed Segment Array* hanya dibentuk pada k > 1 sehingga pemberian nilai *kPrefRight* atau *KO*[*SA*[*i*]][1] dibedakan untuk *k* = 1 pada baris 6 hingga baris 7 dan untuk *k* ≠ 1 pada baris 8 hingga baris 9.

```
/***
* S = string input B
* k = jumlah occurrence yang diinginkan
* n = |S|
* SA = Suffix Array  dari string S
* LCP = larik LCP untuk Suffix Array  S
* KO = Larik k-Occurrence
***/
1.  If(k>1) BuildFixedSegmentArray(k, n, LCP, prev, next)
2.  for i ← 0 to n
3.      if(i+k-1 > n) KO[SA[i]][1] = KO[SA[i]][0] = -1,continue
4.      if(k==1) KO[SA[i]][1] = n-SA[i]
5.      else KO[SA[i]][1] = min(next[i+1],prev[i+k-1])
6.      KO[SA[i]][0] = max(LCP[i], LCP[i+k])+1
7.      if(KO[SA[i]][0] > KO[SA[i]][1])
8.          KO[SA[i]][0] = KO[SA[i]][0] = -1
```

Gambar 4.4 *Pseudocode* Konstruksi larik k-Occurrence

Pada larik *k-Occurrence* nilai *KO*[*SA*[*i*]][0] dan *KO*[*SA*[*i*]][1] masing-masing merepresentasikan nilai *kPrefLeft* dan *kPrefRight*. Seperti yang telah dijelaskan pada Subbab 2.6 informasi *kPrefRight* didapat dari $\min_{i<y\leq i+k-1}\{LCP[y]\}$. Apabila nilai *kPrefRight* > 0 berarti setidaknya terdapat *kPrefRight* karakter pertama pada *suf*(*S*, *i*) yang berulang setidaknya *k* kali. Untuk *k* = 1 tidak diperlukan informasi *kPrefRight* dari $\min_{i<y\leq i+k-1}\{LCP[y]\}$ karena setidaknya *suf*(*S*, *i*) berulang 1 kali hingga nilai *kPrefRight* untuk *k = 1* dapat diketahui dengan panjang dari *suf*(*S*, *i*). Dapat dilihat pada baris 1 struktur *Fixed Segment Array* hanya dibentuk pada k > 1 sehingga pemberian nilai *kPrefRight* atau *KO*[*SA*[*i*]][1] dibedakan untuk *k* = 1 pada baris 6 hingga baris 7 dan untuk *k* ≠ 1 pada baris 8 hingga baris 9.

Nilai *KO*[*SA*[*i*]][0] atau *kPrefLeft* dapat diketahui dengan mencari nilai maksimum dari *lcp*(*SA*[i], *SA*[i-1])+1 dan *lcp*(*SA*[*i*+*k*-1], *SA*[*i*+*k*])+1 (baris 10). Saat nilai *kPrefLeft* > 0 setidaknya terdapat *kPrefLeft* karakter pertama yang mungkin muncul di antara *suf*(*S*, *SA*[*x*]) dimana $i \leq x < i+k$. Maka apabila nilai *kPrefLeft* > *kPrefRight* menunjukkan *kPrefRight* karakter pertama pada *suf*(*S*, *SA*[*i*]) tidak tepat berulang sebanyak *k* kali (baris 13 dan baris 14) dan apabila sebaliknya nilai *kPrefLeft* ditambahkan 1 (baris 12). Untuk nilai *i*+*k* > *n* kita langsung dapat memberi nilai *kPrefLeft* = *kPrefRight* = -1 (baris 3 dan 4) karena saat *i*+*k* > *n* maka tidak mungkin ada kombinasi *prefix* dari *suf*(*S*, *SA*[*i*]) yang berulang setidaknya *k* kali.

3.2.4 Desain Konstruksi Larik *Common Occurrence*

Larik *Common Occurrence* adalah sebuah informasi yang bisa didapat dengan hasil perhitungan menggunakan informasi *Suffix Array* dan LCP pada *Suffix Array tersebut* seperti yang telah

dijelaskan pada Subbab 2.7. *Pseudocode* Konstruksi larik *Common Occurrence* ditunjukkan dalam **Gambar 4.5**.

```
/***
* A = string input A
* C = string hasil concatenation(A,'%',B)
* n = |C|
* SA = Suffix Array  dari string C
* LCP = larik LCP untuk Suffix Array C
* CO = larik Common Occurrence
***/
1.  Nstr1 = length(A)
2.  flag = false
3.  for i ←2 to n
4.      if(SA[i] > Nstr1)
5.          if(!flag)
6.              flag = true
7.              tmp = LCP [i]
8.          else
9.              tmp = min(tmp, LCP[i])
10.         CO[SA[i]-Nstr1-1] = tmp
11.     else flag = false
12. flag = false
13.
14. for i ←n to 2
15.     if(SA[i] > Nstr1)
16.         if(!flag)
17.             flag = true
18.             tmp = LCP [i+1]
19.         else
20.             tmp = min(tmp, LCP [i+1])
21.         CO[SA[i]-Nstr1-1] = max(tmp, CO[SA[i]-Nstr1-1])
22.     else flag = false
```

Gambar 4.5 *Pseudocode* Konstruksi Larik Common Occurrence

Nilai yang tersimpan pada $CO[i]$ adalah batas maksimum *prefix* yang muncul pada *string A* dan *string B* pada $suf(B, i)$ atau bisa dikatakan untuk setiap $pre(suf(B, i), x)$ dimana $0 < x \leq CO[i]$ muncul pada *string A* dan *B*. Nilai $CO[i]$ bisa ditemukan dengan bantuan informasi *Suffix Array* dan LCP dari *string* hasil gabungan *A, B* dimana $CO[SA[i]\text{-}|A|] = \max(lcp(SA[left],SA[i]), lcp(SA[i],SA[right]))$ sesuai dengan penjelasan pada Subbab 2.7. Untuk mengoptimalkan pencarian nilai $CO[SA[i]\text{-}|A|]$ dapat dengan mencari nilai $lcp(SA[left],SA[i])$ untuk semua $CO[SA[i]\text{-}|A|]$ (baris 4 hingga baris 13) lalu dilanjutkan pencarian nilai $lcp(SA[i],SA[right])$ untuk semua $CO[SA[i]\text{-}|A|]$ (baris 17 hingga baris 26).

3.2.5 Desain Proses Kalkulasi Jawaban

Proses kalkulasi dilakukan untuk mencari jawaban dari persoalan yang telah dijelaskan pada Subbab 2.9.1 dengan menggunakan informasi *k-Occurrence* dan *Common Occurrence*. Strategi perhitungan jawaban dengan menggunakan informasi *k-Occurrence* dan *Common Occurrence* dapat dilihat pada Subbab 2.9.2. **Gambar 4.6**. Menunjukkan *pseudocode* untuk proses perhitungan jawaban.

Setelah diketahui informasi *k-Occurrence* dan *Common Occurrence,* jumlah *substring* unik yang sesuai dengan deskripsi soal dapat dihitung dengan menelusuri larik *k-Occurrence* dan larik *Common Occurrence*. Apabila nilai $KO[i][0] \neq -1$, berarti terdapat *prefix* yang berulang sebanyak tepat *k* kali pada batas $KO[i][0]$ hingga $KO[i][1]$. $CO[i] > 0$, menandakan terdapat *prefix* dari $suf(S, i)$ dengan panjang maksimal $CO[i]$ yang muncul pada *string A* dan *string B*. Untuk mencari berapa banyak *substring* unik yang memenuhi deskripsi permasalahan dapat dengan mencari jumlah dari $\min(CO[i], KO[i][1]) - KO[i][0] + 1$ untuk setiap $KO[i][0] \neq -1$ dan $CO[i] >= KO[i][0]$ (baris 6 hingga 9).

```
/***
 * KO = Larik k-Occurrence
 * CO = Larik Common Occurrence
 ***/
1.  ans = 0
2.  for i ← 0 to n-1
3.      kPreftLeft = KO[i][0]
4.      kPrefRight = KO[i][1]
5.      matchChar = CO[i]
6.      if(kPreftLeft!= -1 && matchChar >= kPreftLeft)
7.      ans += min(matchChar, kPrefRight) - kPreftLeft + 1
```

Gambar 4.6 *Pseudocode* Proses Kalkulasi Jawab

(Halaman ini sengaja dikosongkan)

BAB IV
IMPLEMENTASI

Pada bab ini dijelaskan mengenai implementasi berdasarkan desain algoritma serta struktur data yang telah dilakukan dalam menyelesaikan permasalahan SPOJ klasik 12713 *Strings*.

4.1 Lingkungan Implementasi

Lingkungan implementasi yang akan digunakan sebagai berikut:

1. Perangkat Keras
   - Processor Intel® Core™ i7-3630QM CPU @2.40GHz -
   - RAM 8.00 GB
2. Perangkat Lunak
   - Operating System Window 8.1
   - Integrated Development Environment Code::Blocks 13.12

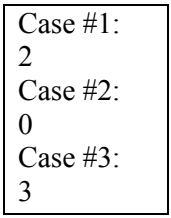
4.2 Data Masukan

Data masukan merupakan data kasus sesuai dengan deskripsi permasalahan yang dijelaskan pada Subbab 2.9.1 dan diproses dalam program yang dibangun menurut desain yang telah dipaparkan pada Subbab 3.1. Pada baris pertama, diberikan nilai sebuah nilai *integer t* yang menentukan berapa banyak kasus yang ingin diproses pada sebuah data *input*. Untuk setiap kasus akan terdiri atas dua *input string A, B* dan sebuah *integer k.*

Gambar 5.1 Contoh Data Masukan Program

4.3 Data Keluaran

Data keluaran yang dihasilkan oleh program untuk setiap data uji adalah *string* `Case#no_data_uji` dan sebuah integer yang mewakili jumlah *substring* unik *A* yang terdapat pada *string B* dan tepat berulang sebanyak *k* kali pada *string B*.



Gambar 5.2 Contoh Data Keluaran Program

4.4 Implementasi Fungsi Main

Fungsi Main diimplementasikan sesuai dengan *Pseudocode* yang ditunjukkan dalam **Gambar 4.1**. Implementasi Fungsi Main ditunjukkan dalam **Kode Sumber 5.1**.

Dalam sistem yang dibangun untuk menyelesaikan penelitian ini diperlukan sebuah fungsi tambahan yang digunakan

untuk menginisialisasi beberapa variabel yang digunakan di operasi selanjutnya.

```
1.      char str1[MAX_N],str2[MAX_N];
2.      int m,Nstr1,ans;
3.      int occ[MAX_N][2],com[MAX_N];
4.      scanf("%d",&t);
5.      for(int tt=1;tt<=t;tt++){
6.              scanf("%s",str1);
7.              scanf("%s",str2);
8.              scanf("%d",&k);
9.              strcpy(T,str2);
10.             n = (int)strlen(T);
11.             buildSuffixArray(T, n, &sa[0]);
12.             computeInverseSuffixArray(&sa[0], n,
    &isa[0]);
13.             computeLCPArray(T, &sa[0], &isa[0], n,
    &lcp[0]);
14.
15.              //konstruksi larik k-Occurrence
16.
17.             //konstruksi larik Common Occurrence
18.
19.            //proses kalkulasi jawaban
20.             printf("Case #%d:\n",tt);
21.             printf("%d\n",ans);
22.
23.     }
24.     return 0;
25. }
```

Kode Sumber 5.1 Implementasi Fungsi Main

Semua proses inisialisasi variabel ditaruh pada fungsi resetData dan implementasi fungsi dapat dilihat pada **Kode Sumber 5.2**

```
1.  void resetData(int x){
2.      memset(T,0,sizeof(T));
3.      for(int i=0;i<x+2;i++){
4.          sa[i] = isa[i] = lcp[i] = 0;
5.      }
6.  }
```

Kode Sumber 5.2 Implementasi Fungsi resetData

4.4 Implementasi Fungsi Konstruksi *Longest Common Prefix*

Fungsi computeLCPArray diperlukan untuk membangun informasi LCP dari sebuah *Suffix Array* yang nantinya digunakan untuk mendapatkan informasi lain yang dibutuhkan dalam menyelesaikan permasalahan yang dikerjakan pada penelitian ini. Seperti mencari informasi *substring* mana saja yang berulang tepat sebanyak *k* kali dan mencari *substring* yang ada pada dua *string* yang diberikan. Implementasi fungsi computeLCPArray ditunjukkan pada **Kode Sumber 5.3** sesuai dengan desain *pseudocode* yang ada pada **Gambar 4.2**

```
1.  static void computeLCPArray(const char *str, const int
    *suffixArray, const int *inverseSuffixArray, int n, int
    *lcpArray) {
2.      int h = 0;
3.      for(Index i = 0; i < n; i ++) {
4.          int pos = inverseSuffixArray[i];
5.          int j = suffixArray[pos-1];
6.          int hbound = min(n - j, n - i);
7.          for(int k = 0; h < hbound && str[i+h] ==
    str[j+h]; ++ h) ;
8.          lcpArray[pos] = h;
9.          if(h > 0) -- h;}
10.     lcpArray[0] = 0;}
```

Kode Sumber 5.3 Implementasi Fungsi computeLCPArray

4.5 Implementasi Fungsi Konstruksi *Fixed Segment Array*

Fungsi buildFixedSegmentArray digunakan untuk membangun struktur data *Fixed Segment Array* yang nanti digunakan dalam konstruksi larik *k-Occurrence*. Implementasi fungsi buildFixedSegmentArray dapat dilihat pada **Kode Sumber 5.4** yang di implementasi sesuai dengan desain *pseudocode* yang ditunjukkan pada **Gambar 4.3**.

```
1.  void buildFixedSegmentArray(int k,int n,int *LCP,int
    *prev,int *next){
2.      int rep = k-1;
3.      int temp = n/rep;
4.      if(n%rep) temp++;
5.      temp = temp*rep;
6.      for(int i=n+1;i<=temp;i++)lcp[i] = 0;
7.      for(int i=0;i<=temp;i++){
8.          if(i%rep==0){
9.              prev[i] = lcp[i];
10.         }
11.         else{
12.             prev[i] = min(prev[i-1],lcp[i]);
13.         }
14.     }
15.     for(int i=dumb;i>=0;i--){
16.         if(i%rep==rep-1)
17.             next[i] = lcp[i];
18.         else
19.             next[i] = min(next[i+1],lcp[i]);
20.     }
21. }
```

Kode Sumber 5.4 Implementasi Fungsi buildFixedSegmentArray

4.6 Implementasi Konstruksi Larik *k-Occurrence*

Implementasi proses konstruksi larik *k-Occurrence* ditunjukkan pada **Kode Sumber 5.5** sesuai dengan *pseudocode* yang ditunjukkan pada **Gambar 4.4**. Pada implementasi konstruksi larik *k-Occurrence* diperlukan informasi *Suffix Array* dan LCP sehingga pada baris 3 hingga baris 5 pada proses konstruksi array dimulai dengan membangun informasi *Suffix Array* dan LCP.

```
1.  strcpy(T,str2);
2.  n = (int)strlen(T);
3.  buildSuffixArray(T, n, &sa[0]);
4.  computeInverseSuffixArray(&sa[0], n, &isa[0]);
5.  computeLCPArray(T, &sa[0], &isa[0], n, &lcp[0]);
6.  if(k>1)
    buildFixedSegmentArray(k,n,&lcp[0],&prev[0],&next[0]);
7.  for(int i=1;i<=n;i++){
8.      if(i+k-1>n){
9.          occ[sa[i]][1] = occ[sa[i]][0] = -1;
10.         continue;
11.     }
12.
13.     if(k==1)
14.         occ[sa[i]][1] = n-sa[i];
15.     else
16.         occ[sa[i]][1] = min(next[i+1],prev[i+k-1]);
17.     m = max(lcp[i], lcp[i+k]);
18.     if(occ[sa[i]][1]>m)
19.         occ[sa[i]][0] = m+1;
20.     else
21.         occ[sa[i]][1] = occ[sa[i]][0] = -1;
22. }
23. resetData(n);
```

Kode Sumber 5.5 Implementasi Konstruksi Larik *k-Occurrence*

4.7 Implementasi Konstruksi Larik *Common Occurrence*

Implementasi proses konstruksi larik *Common Occurrence* ditunjukkan pada **Kode Sumber 5.6** dan **Kode Sumber 5.7** sesuai dengan *pseudocode* yang ditunjukkan pada **Gambar 4.5**. Dalam proses konstruksi larik *Common Occurrence* diperlukan *string* baru yang merupakan hasil *concatenation* dari *string A* dan *string B* seperti yang ditunjukkan pada baris 1 hingga baris 3 (seperti yang dijelaskan pada Subbab 2.7). Setelah *string* baru tersebut didapat baru dibentuk informasi *Suffix Array* beserta LCP yang nantinya dipakai untuk proses konstruksi larik *Common Occurrence*.

```
1.  strcpy(T,str1);
2.  strcat(T,"/");
3.  strcat(T,str2);
4.  n = (int)strlen(T);
5.  buildSuffixArray(T, n, &sa[0]);
6.  computeInverseSuffixArray(&sa[0], n, &isa[0]);
7.  computeLCPArray(T, &sa[0], &isa[0], n, &lcp[0]);
8.  int Nstr1 = strlen(str1);
9.  bool tes = false; int tmp;
10. for(int i=2;i<=n;i++){
11.       if(sa[i]>Nstr1){
12.                 if(!tes){
13.                       tes = true;
14.                       tmp = lcp[i];
15.                  }
16.                 else tmp = min(tmp,lcp[i]);
17.                 com[sa[i]-Nstr1-1] = tmp;
18.       }
19.       else tes = false;
20. }
21. bool tes = false;
```

Kode Sumber 5.6 Implementasi Konstruksi Larik *Common Occurrence* (a)

```
22. for(int i=n;i>=2;i--){
23.       if(sa[i]>Nstr1){
24.            if(!tes){
25.                 tes = true;
26.                 tmp = lcp[i+1];
27.            }
28.            else
29.                 tmp = min(tmp,lcp[i+1]);
30.
31.            com[sa[i]-Nstr1-1]  =  max(tmp,com[sa[i]-Nstr1-
    1]);
32.       }
33.       else tes = false;
34. }
35. resetData(n);
```

Kode Sumber 5.7 Implementasi Konstruksi Larik *Common Occurrence* (b)

4.8 Implementasi Proses Kalkulasi Jawaban

Implementasi proses kalkulasi jawaban dari informasi *Common Occurrence* dan k-*Occurrence* yang telah diketahui, ditunjukkan pada **Kode Sumber 5.8** sesuai dengan *pseudocode* pada **Gambar 4.6**.

```
1.  ans=0;
2.  n = strlen(str2);
3.  for (int i = 0; i < n; i++) {
4.       kPrefLeft = occ[i][0];
5.       matchChar = com[i];
6.       kPrefRight= occ[i][1];
7.       if(kPrefLeft>-1 && matchChar >= kPrefLeft){
8.            ans += min(matchChar, kPrefRight) - kPrefLeft + 1;
9.       }
10. }
```

Kode Sumber 5.8 Implementasi Proses Kalkulasi Jawaban

BAB V
UJI COBA DAN EVALUASI

Pada bab ini akan dijelaskan uji coba yang dilakukan pada sistem yang telah dikerjakan serta analisis dari uji coba yang telah dilakukan. Pembahasan pengujian meliputi lingkungan uji coba, skenario uji coba yang meliputi uji kebenaran dan uji kinerja serta analisis setiap pengujian.

5.1 Lingkungan Uji Coba

Lingkungan uji coba yang akan digunakan sebagai berikut:

1. Perangkat Keras
   - Processor Intel® Core™ i7-3630QM CPU @2.40GHz -
   - RAM 8.00 GB
2. Perangkat Lunak
   - Operating System Window 8.1
   - Integrated Development Environment Code::Blocks 13.12

5.2 Skenario Uji Coba

Uji coba ini dilakukan untuk menguji apakah sistem telah diimplementasikan dengan benar dan berjalan sebagaimana mestinya. Uji coba akan didasarkan pada beberapa skenario untuk menguji kesesuaian dan kinerja aplikasi.

5.2.1 Uji Coba Kebenaran Suffix Array

Untuk menguji kebenaran *Suffix Array* yang digunakan pada penelitian ini. *Suffix Array* yang digunakan diuji dengan menggunakan persoalan parsial 6409 Suffix Array (**Gambar A.4**)

pada situs SPOJ. Dimana dalam persoalan tersebut diminta untuk mengeluarkan *Suffix Array* yang dibangun dari sebuah *string* yang diberikan. Maksimal *string* yang dimasukkan tidak lebih dari 100.000 karakter dengan batas waktu 0.050 detik. Hasil uji coba *Suffix Array* pada situs SPOJ ditunjukkan dalam **Gambar 5.1**.

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
14176503		2015-04-30 10:50:33	Suffix Array	100 edit run	0.04	2.9M	C++ 4.3.2

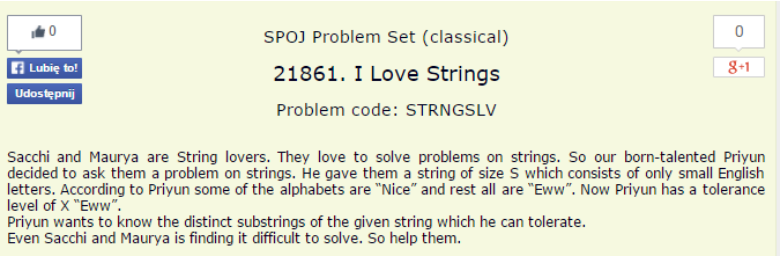
Gambar 5.1Hasil Uji Coba Suffix Array pada situs SPOJ

Dari hasil uji coba yang telah dilakukan, kode sumber program yang dikirimkan mendapat umpan balik ***Accepted*** dengan nilai 100. Waktu yang dibutuhkan program adalah 0,04 detik dan memori yang dibutuhkan program adalah 2,9 MB. Hal tersebut membuktikan bahwa implementasi *Suffix Array* yang digunakan telah benar dan optimal. Dimana penilaian dalam persoalan Suffix Array pada situs SPOJ dibagi ke dalam beberapa kelompok berikut sesuai dengan kompleksitas pembentukan *Suffix Array*:

1. $O(n^2 \log(n))$ mendapat nilai 20-30
2. $O(n \log^2(n))$ mendapat nilai 40
3. $O(n \log(n))$ mendapat nilai 60-70
4. $O(n)$ dengan implementasi tidak optimal mendapat nilai 80-90
5. $O(n)$ dengan implementasi optimal mendapat nilai 100

5.2.2 Uji Coba Kebenaran *Longest Common Prefix*

Untuk menguji kebenaran implementasi *Longest Common Prefix* pada *Suffix Array* yang digunakan pada penelitian ini, digunakan persoalan klasik 21861. *I Love Strings* pada situs SPOJ. Deskripsi permasalahan ditunjukkan dalam **Gambar 5.2**.



Gambar 5.2 Deskripsi Permasalahan I Love Strings

Deskripsi singkat permasalahan tersebut adalah diberikan sebuah *string S*, sebuah himpunan karakter *C*, dan sebuah *integer X*. Hitung berapa banyak *substring* unik dari *S* dimana dalam *substring* tersebut jumlah karakter yang tidak terdapat dalam himpunan *C* harus kurang dari *X*. Berikut merupakan format masukan dari permasalahan tersebut:

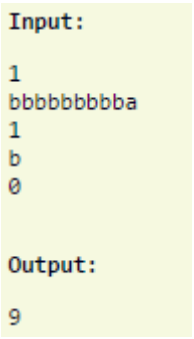
1. Masukan terdiri atas beberapa kasus uji, format masukan setiap kasus uji didefinisikan pada poin hingga 2 poin 5.
2. Baris pertama berisi sebuah *string S* yang terdiri atas huruf alfabet kecil.
3. Baris kedua berisi sebuah *integer P* yang merepresentasikan jumlah elemen pada himpunan *C*.
4. *P* baris berikutnya berisi 1 buah karakter huruf alfabet kecil yang merepresentasikan elemen dalam himpunan *C*.
5. Pada akhir baris untuk setiap kasus uji akan berisi sebuah *integer X* yang merepresentasikan batas karakter yang tidak terdapat dalam himpunan *C* untuk setiap *substring*.

Untuk setiap kasus uji keluarkan sebuah *integer N* yang merepresentasikan jumlah *substring* unik dari *string S* yang sesuai dengan deskripsi permasalahan.

Berikut merupakan batasan dari permasalahan tersebut:

1. $1 \leq |S| \leq 2.000$
2. $0 \leq P \leq 26$
3. $0 \leq X \leq 2.000$

Contoh masukan dan keluaran dari permasalahan tersebut ditunjukkan dalam **Gambar 5.3**.



Gambar 5.3Contoh Masukan dan Keluaran Permasalahan I Love Strings

Suffix Array yang dilengkapi dengan informasi *Longest Common Prefix* dapat digunakan untuk menyelesaikan permasalahan I Love String, terutama saat perhitungan jumlah *substring* unik yang dapat dibentuk dalam sebuah *string*. Untuk menemukan jumlah *substring* unik dalam sebuah *string S* dapat menggunakan informasi LCP dari *Suffix Array* yang dibentuk dari *string* itu sendiri. Saat nilai $n > 0$ dimana $n = LCP[i]$, maka semua $pre(suf(S, SA[i]), x)$ untuk $0 < x \leq n$ dan $0 < i \leq |S|$ sudah muncul pada *suffix* sebelumnya di pada indeks *Suffix Array* yang lebih kecil. Pada **Gambar 5.4** dapat dilihat nilai $LCP[4] > 0$, sehingga *prefix-prefix* dari $pre(suf(S,SA[4]),1)$ hingga $pre(suf(S,SA[4]),4)$ (kolom dengan warna krem) pernah muncul di indeks *Suffix Array* sebelumnya (berlaku juga dengan $LCP[5]$ dan seterusnya).

i	*SA*[i]	*LCP*[i]	Suf(*S*, *SA*[i])													
0	14	0	$													
1	0	0	a	b	c	d	b	c	d	e	f	b	c	d	b	d	$
2	1	0	b	c	d	b	c	d	e	f	b	c	d	b	d	$	
3	9	4	b	c	d	b	d	$									
4	4	3	b	c	d	e	f	b	c	d	b	d	$				
5	12	1	b	d	$												
6	2	0	c	d	b	c	d	e	f	b	c	d	b	d	$		
7	10	3	c	d	b	d	$										
8	5	2	c	d	e	f	b	c	d	b	d	$					
9	13	0	d	$													
10	3	1	d	b	c	d	e	f	b	c	d	b	d	$			
11	11	2	d	b	d	$											
12	6	1	d	e	f	b	c	d	b	d	$						
13	7	0	e	f	b	c	d	b	d	$							
14	8	0	f	b	c	d	b	d	$								

Gambar 5.4 Contoh Larik *SA* dan *LCP* pada *String* `abcdbcdefbcdbd$`

Sehingga dalam permasalahan *I Love String substring-substring* yang dicek hanya *prefix-prefix* dari *pre*(*suf*(*S*, *SA*[*i*]), *y*) untuk *LCP*[*i*] < *y* < |*S*| - *SA*[*i*]. Hasil uji coba implementasi *Longest Common Prefix* pada *Suffix Array* pada situs SPOJ ditunjukkan dalam **Gambar 5.5**.

13728637		2015-02-25 05:25:26	I Love Strings	accepted edit run	0.00	2.8M	C++ 4.3.2

Gambar 5.5 Hasil Uji Coba *Longest Common Prefix* pada *Suffix Array* di Situs SPOJ

Dari hasil uji coba yang telah dilakukan, kode sumber program yang dikirimkan mendapat umpan balik ***Accepted***. Waktu yang dibutuhkan program adalah 0,00 detik dan memori yang dibutuhkan program adalah 2,8 MB. Hal tersebut membuktikan

bahwa implementasi *Longest Common Prefix* pada *Suffix Array* yang digunakan telah benar.

5.2.3 Simulasi Penyelesaian Permasalahan Klasik 21861 *I Love Strings*

Untuk memudahkan dalam memahami solusi yang digunakan untuk menyelesaikan permasalahan *I Love String.* Pada Subbab 5.2.3 akan menjelaskan tentang langkah-langkah dalam menyelesaikan permasalahan *I Love String*. **Gambar 5.6** merupakan kasus uji yang digunakan dalam proses simulasi.

```
1
abcdebcd
3
a
c
e
1
```

Gambar 5.6 Contoh Kasus Uji *I Love Strings*

Untuk dapat membedakan huruf *nice* atau tidak kita dapat membuat sebuah larik *flag* dimana *flag*[*i*] = 0 apabila *S*[*i*] merupakan huruf *nice* dan *flag*[*i*] = 1 apabila *S*[*i*] merupakan huruf *eww*. **Gambar 5.7** menunjukkan *Suffix Array* dan LCP yang dibentuk dari *string* masukan.

Apabila karakter-karakter yang terdapat dalam *string* masukan diganti dengan nilai *flag* setiap karakter maka informasi *suffix-suffix* yang ada dalam *Suffix Array* dapat dilihat seperti **Gambar 5.8**. Dengan mengganti karakter-karakter *string* dengan nilai *flag,* kita dapat mengetahui berapa banyak karakter *eww* yang ada pada suatu *substring S*[*a*]…*S*[*b*] dimana $0 \leq a \leq b < |S|$ dengan menjumlahkan elemen-elemen pada *flag*[*a*] hingga *flag*[*b*].

i	*SA*[*i*]	*LCP*[*i*]	suf(*S*,*i*)								
0	8	0	$								
1	0	0	a	b	c	d	e	b	c	d	$
2	5	0	b	c	d	$					
3	1	3	b	c	d	e	b	c	d	$	
4	6	0	c	d	$						
5	2	2	c	d	e	b	c	d	$		
6	7	0	d	$							
7	3	1	d	e	b	c	d	$			
8	4	0	e	b	c	d	$				

Gambar 5.7 Larik *SA*, *LCP* dari S*tring* `abcdebcd$`

i	*SA*[*i*]	*LCP*[*i*]	suf(*S*,*i*)								
0	8	0	$								
1	0	0	0	1	0	1	0	1	0	1	$
2	5	0	1	0	1	$					
3	1	3	1	0	1	0	1	0	1	$	
4	6	0	0	1	$						
5	2	2	0	1	0	1	0	1	$		
6	7	0	1	$							
7	3	1	1	0	1	0	1	$			
8	4	0	0	1	0	1	$				

Gambar 5.8 Larik *SA*, *LCP* dari S*tring* `abcdebcd$` dimana Nilai Setiap Karakter diganti Dengan Nilai *flag*[*i*]

Seperti yang dijelaskan pada Subbab 5.2.2 *prefix* yang dihitung hanya *prefix* pada *pre*(*suf*(*S*,*SA*[*i*]),*y*) untuk $LCP[i] < y < |S| - SA[i]$ karena *pre*(*suf*(*S*,*SA*[*i*]),*y*) untuk $0 < y \leq LCP[i]$ dan $0 < i \leq |S|$ pasti telah dihitung sebelumnya. Sehingga untuk setiap indeks *i* perlu dicari sebuah bilangan *z* dimana *z* adalah indeks terbesar dimana jumlah *flag*[*SA*[*i*]] …. *flag*[*z*] masih kurang dari nilai *X* (batas maksimum jumlah karakter *eww* pada suatu *substring*) apabila tidak ada nilai *z* yang memenuhi maka *z* = -1.

Nilai *validPref* merepresentasikan *prefix* unik yang memenuhi syarat dimana *validPref* = min(*z*-(*SA*[*i*]+*LCP*[*i*])+1, 0). **Gambar 5.9** menunjukkan hasil perhitungan nilai *z* untuk setiap indeks *i* dan total penjumlahan dari semua nilai *validPref*. **Gambar A.3** menunjukkan *prefix* yang direpresentasikan oleh nilai *validPref*.

i	*SA*[*i*]	*LCP*[*i*]	*z*	*validPref*
1	0	0	2	3
2	5	0	6	2
3	1	3	2	0
4	6	0	7	2
5	2	2	4	1
6	7	0	7	1
7	3	1	4	1
8	4	0	6	3

SUM *validPref* = 13

Gambar 5.9 Hasil Perhitungan Solusi *I Love Strings*

5.2.4 Simulasi Penyelesaian Permasalahan Klasik 12713 *Strings*

Untuk membuktikan kebenaran strategi yang dilakukan, maka pada Subbab ini akan dicoba mensimulasikan strategi yang dijelaskan pada Subbab 2.9.2 dalam kasus kecil.

```
fastlast
bestmost
2
```

Gambar 5.10 Contoh Kasus Uji *Strings*

Dari **Gambar 5.10** dapat diketahui nilai *A* = `fastlast$`, *B* = `bestmost$`, *k* = 2. Larik *KO* yang dibentuk oleh *string B* dan larik *CO* yang dibentuk oleh *string* gabungan *A* & *B* ditunjukkan pada **Gambar 5.11**.

i	0	1	2	3	4	5	6	7
KO[*i*][0]	-1	-1	-1	-1	-1	-1	1	1
KO[*i*][1]	-1	-1	-1	-1	-1	-1	2	1
CO[*i*]	0	0	2	1	0	0	2	1

Gambar 5.11 Larik *KO* & *CO* yang dibentuk dari *String* Masukan

Dari larik *KO*[6] bisa dilihat bahwa *pre*(*suf*(*B*,6),*y*) dimana $KO[6][0] \leq y \leq KO[6][1]$ berulang tepat sebanyak 2 kali pada *string B* sehingga larik *KO* telah merepresentasikan *prefix-prefix* mana saja yang berulang tepat sebanyak *k* kali dan dari larik *CO* dapat diketahui untuk setiap $CO[i] \neq 0$ berarti *substring pre*(*suf*(*B*, *i*), *z*) dimana $0 < z \leq CO[i]$ muncul pada *string A* & *B*. Oleh karena itu saat dilakukan proses kalkulasi jawaban mendapatkan hasil 3 yang berasal dari perhitungan pada indeks 6 & 7. Hasil tersebut sesuai dengan contoh keluaran yang terdapat dalam deskripsi soal **Gambar 5.12**.

```
Input:
3
egyptnational
ecpc
1
egyptnational
ecpc
2
fastlast
bestmost
2

Output:
Case #1:
2
Case #2:
0
Case #3:
3
```

Gambar 5.12 Contoh Masukan dan Keluaran dalam Deskripsi Permasalahan *Strings*

5.2.5 Uji Coba Kebenaran *Suffix Array & Longest Common Prefix* untuk Permasalahan SPOJ Klasik 12713 *Strings*

Uji coba kebenaran dilakukan dengan mengirim kode sumber implementasi yang telah dilakukan ke situs SPOJ. Permasalahan yang akan diselesaikan adalah *Strings* seperti yang dijelaskan pada Subbab 2.9.2. Dimana dalam persoalan ini akan diuji kebenaran informasi *Suffix Array, Longest Common Prefix, K-Occurrence,* dan *Common Occurrence.* Setelah kode sumber implementasi dikirimkan ke situs SPOJ, dapat dilihat umpan balik sistem pada situs SPOJ seperti yang dijelaskan pada Subbab 2.8. Hasil uji coba pada situs SPOJ ditunjukkan dalam **Gambar 5.13**.

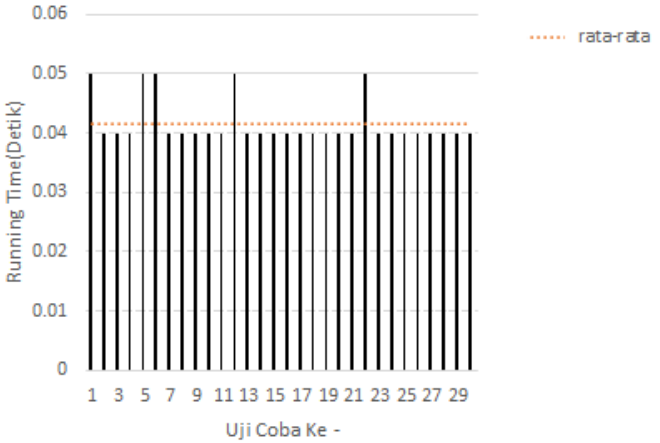
2015-04-30	11:02:51	Strings	Accepted	0.04	3.2M	C++ 4.3.2

Gambar 5.13 Hasil Uji Coba permasalahan *Strings* pada situs SPOJ

Dari hasil uji coba yang telah dilakukan, kode sumber program yang dikirimkan mendapat umpan balik ***Accepted***. Waktu yang dibutuhkan program adalah 0,04 detik dan memori yang dibutuhkan program adalah 3,2 MB. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan berhasil menyelesaikan permasalahan *multiple string matching* dengan menggunakan *Suffix Array* yang dilengkapi dengan informasi *Longest Common Prefix*.

Setelah itu dilakukan pengiriman kode sumber implementasi sebanyak 30 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Hasil uji coba pada situs SPOJ sebanyak 30 kali ditunjukkan dalam **Tabel A.1, Tabel A.2**, **Gambar 5.14**. Dari hasil uji coba yang telah dilakukan, seluruh kode sumber program yang dikirimkan mendapat umpan balik ***Accepted***. *Running time* yang dibutuhkan program setelah dilakukan uji coba didapatkan waktu minimum sebesar 0,04 detik, maksimum sebesar 0,05 detik, dan rata-rata sebesar 0,042 detik

yang digambarkan oleh garis jingga pada **Gambar 5.14**. Memori yang dibutuhkan program tetap 3.2 MB.



Gambar 5.14 Grafik Hasil Uji Coba pada Situs SPOJ Sebanyak 30 Kali

(Halaman ini sengaja dikosongkan)

BAB VI
KESIMPULAN

Pada bab ini penulis menjelaskan tentang kesimpulan dari hasil uji coba yang telah dilakukan dan saran mengenai hal-hal yang masih bisa dikembangkan.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan *multiple string matching* dapat diambil kesimpulan sebagai berikut:

1. Permasalahan *multiple string matching* yang diselesaikan menggunakan *Suffix Array* yang dilengkapi informasi *Longest Common Prefix* memiliki kompleksitas waktu O(*n*). Artinya waktu yang dibutuhkan program dipengaruhi oleh panjang *string* secara linear.
2. Dari uji coba yang dilakukan implementasi *Suffix Array, Longest Common Prefix, K-Occurrence, dan Common Occurrence* sudah menghasilkan hasil yang diharapkan.
3. Waktu rata-rata yang diperlukan untuk menyelesaikan permasalahan *multiple string matching* adalah sebesar 0.042 detik.
4. Informasi *Longest Common Prefix* pada *Suffix Array* dapat dimanfaatkan untuk mencari sebuah substring terjadi berapa kali pada suatu string, mencari substring unik yang dibentuk dari sebuah string, dan mencari substring apa saja yang merupakan anggota dari dua buah string.

(Halaman ini sengaja dikosongkan)

DAFTAR PUSTAKA

[1] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast Pattern Matching in String," *SICOMP,* vol. 6, pp. 323-350, 1977.

[2] R. M. Karp and M. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development,* vol. XXXI, no. 2, 1987.

[3] G. Nong, S. Zhang and W. H. Chan, "Linear Suffix Array Construction by Almost Pure Induced-Sorting".

[4] "Sphere Online Judge," Sphere Research Lab, [Online]. Available: http://www.spoj.com/.

[5] M. F. Aziz, IMPLEMENTASI DAN PERBANDINGAN SUFFIX ARRAY STATIS DAN DINAMIS UNTUNG STRING INPUT YANG MENGALAMI PEMBARUAN, Surabaya, 2013.

[6] S. Halim and F. Halim, Competitive Programming 3: The New Lower Bound of Programming Contests, 2013.

[7] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," 1989.

(Halaman ini sengaja dikosongkan)

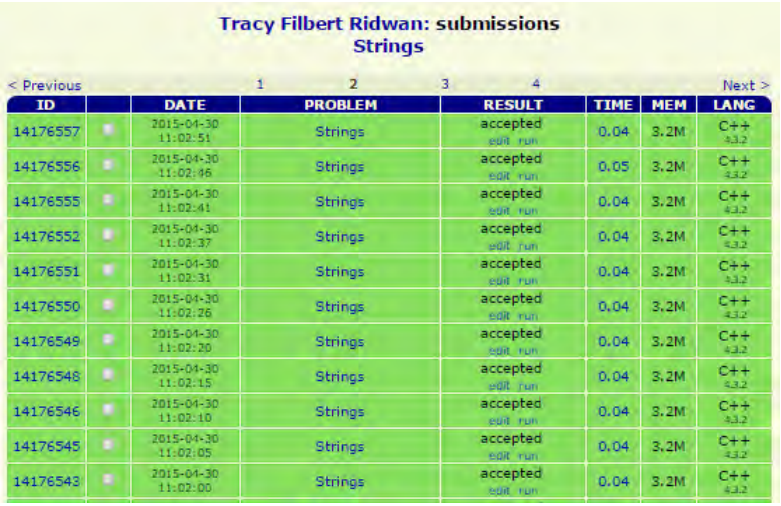
LAMPIRAN A

Tabel A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali (1)

No	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	0.05	3.2
2	Accepted	0.04	3.2
3	Accepted	0.04	3.2
4	Accepted	0.04	3.2
5	Accepted	0.05	3.2
6	Accepted	0.05	3.2
7	Accepted	0.04	3.2
8	Accepted	0.04	3.2
9	Accepted	0.04	3.2
10	Accepted	0.04	3.2
11	Accepted	0.04	3.2
12	Accepted	0.05	3.2
13	Accepted	0.04	3.2
14	Accepted	0.04	3.2
15	Accepted	0.04	3.2
16	Accepted	0.04	3.2
17	Accepted	0.04	3.2
18	Accepted	0.04	3.2
19	Accepted	0.04	3.2
20	Accepted	0.04	3.2
21	Accepted	0.04	3.2
22	Accepted	0.05	3.2
23	Accepted	0.04	3.2
24	Accepted	0.04	3.2

Tabel A.2 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali (2)

No	Hasil	Waktu (detik)	Memori (MB)
25	Accepted	0.04	3.2
26	Accepted	0.04	3.2
27	Accepted	0.04	3.2
28	Accepted	0.04	3.2
29	Accepted	0.04	3.2
30	Accepted	0.04	3.2

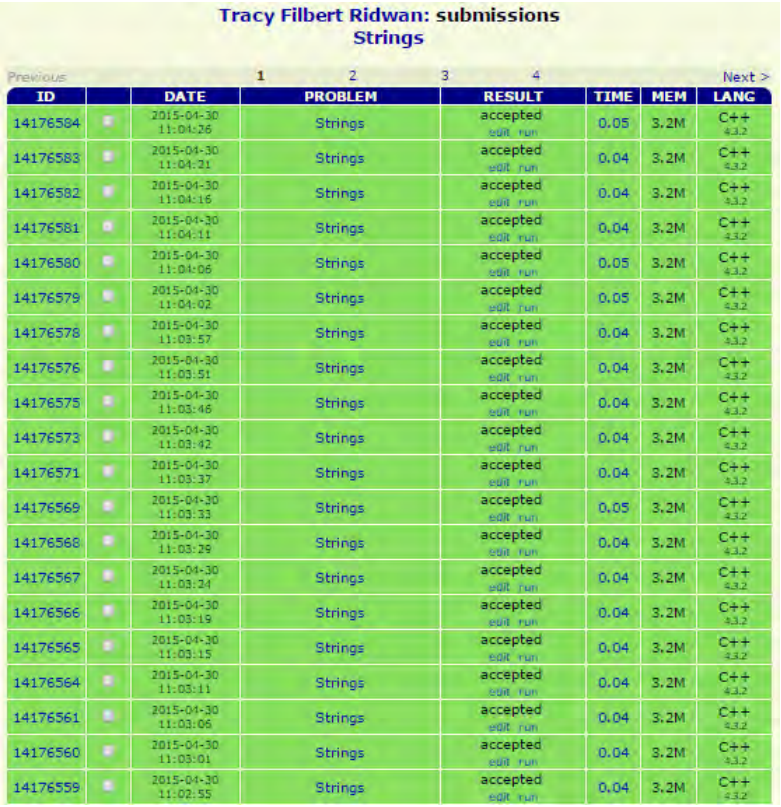


Tracy Filbert Ridwan: submissions
Strings

< Previous 1 2 3 4 Next >

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
14176557		2015-04-30 11:02:51	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176556		2015-04-30 11:02:46	Strings	accepted edit run	0.05	3.2M	C++ 4.3.2
14176555		2015-04-30 11:02:41	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176552		2015-04-30 11:02:37	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176551		2015-04-30 11:02:31	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176550		2015-04-30 11:02:26	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176549		2015-04-30 11:02:20	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176548		2015-04-30 11:02:15	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176546		2015-04-30 11:02:10	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176545		2015-04-30 11:02:05	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176543		2015-04-30 11:02:00	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2

Gambar A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali (1)



Tracy Filbert Ridwan: submissions
Strings

Previous 1 2 3 4 Next >

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
14176584		2015-04-30 11:04:26	Strings	accepted edit run	0.05	3.2M	C++ 4.3.2
14176583		2015-04-30 11:04:21	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176582		2015-04-30 11:04:16	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176581		2015-04-30 11:04:11	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176580		2015-04-30 11:04:06	Strings	accepted edit run	0.05	3.2M	C++ 4.3.2
14176579		2015-04-30 11:04:02	Strings	accepted edit run	0.05	3.2M	C++ 4.3.2
14176578		2015-04-30 11:03:57	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176576		2015-04-30 11:03:51	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176575		2015-04-30 11:03:46	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176573		2015-04-30 11:03:42	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176571		2015-04-30 11:03:37	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176569		2015-04-30 11:03:33	Strings	accepted edit run	0.05	3.2M	C++ 4.3.2
14176568		2015-04-30 11:03:29	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176567		2015-04-30 11:03:24	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176566		2015-04-30 11:03:19	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176565		2015-04-30 11:03:15	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176564		2015-04-30 11:03:11	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176561		2015-04-30 11:03:06	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176560		2015-04-30 11:03:01	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2
14176559		2015-04-30 11:02:55	Strings	accepted edit run	0.04	3.2M	C++ 4.3.2

Gambar A.2 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali (2)

1. i = 1, SA[i] = 0, LCP[i] = 0 ,*suf*(*S*,*i*) = abcdebcd$
   - a
   - ab
   - abc
2. i = 2, SA[i] = 5, LCP[i] = 0 ,*suf*(*S*,*i*) = bcd$
   - b
   - bc
3. i = 3, SA[i] = 1, LCP[i] = 3 ,*suf*(*S*,*i*) = bcdebcd$
   - b
   - bc
4. i = 4, SA[i] = 6, LCP[i] = 0 ,*suf*(*S*,*i*) = cd$
   - c
   - cd
5. i = 5, SA[i] = 2, LCP[i] = 2 ,*suf*(*S*,*i*) = cdebcd$
   - c
   - cd
   - cde
6. i = 6, SA[i] = 7, LCP[i] = 0 ,*suf*(*S*,*i*) = d$
   - d
7. i = 7, SA[i] = 3, LCP[i] = 1 ,*suf*(*S*,*i*) = debcd$
   - d
   - de
8. i = 8, SA[i] = 4, LCP[i] = 0 ,*suf*(*S*,*i*) = ebcd$
   - e
   - eb
   - ebc

prefix yang berwarna merah menandakan *prefix* yang pernah berulang dan jumlah prefix yang berwarna hitam pada setiap *i* sama dengan nilai *validPref* pada setiap *i*. *Prefix* yang dicantumkan hanyalah *prefix* yang mempunyai karakter *eww* lebih kecil dari *X*

Gambar A.3 Representasi nilai *validPref*

SPOJ Problem Set (partial)

6409. Suffix Array

Problem code: SARRAY

Given a string of length at most 100,000 consist of alphabets and numbers. Output the suffix array of the string.

A suffix array is an array of integers giving the starting positions (0-based) of suffixes of a string in lexicographical order. Consider a string "abracadabra0AbRa4Cad14abra". The size of the suffix array is equal to the length of the string. Below is the list of 26 suffixes of the string along with its starting position sorted in lexicographical order:

```
POS SUFFIX
 11 0AbRa4Cad14abra
 20 14abra
 16 4Cad14abra
 21 4abra
 12 AbRa4Cad14abra
 17 Cad14abra
 14 Ra4Cad14abra
 25 a
 10 a0AbRa4Cad14abra
 15 a4Cad14abra
 22 abra
  7 abra0AbRa4Cad14abra
  0 abracadabra0AbRa4Cad14abra
  3 acadabra0AbRa4Cad14abra
 18 ad14abra
  5 adabra0AbRa4Cad14abra
 13 bRa4Cad14abra
 23 bra
  8 bra0AbRa4Cad14abra
  1 bracadabra0AbRa4Cad14abra
  4 cadabra0AbRa4Cad14abra
 19 d14abra
  6 dabra0AbRa4Cad14abra
 24 ra
  9 ra0AbRa4Cad14abra
  2 racadabra0AbRa4Cad14abra
```

Note: this is a partial score problem.
$O(n^2 \log(n))$ is expected to score about 20-30. (Naive sorting all suffixes)
$O(n \log^2(n))$ is expected to score about 40. (OK for most programming contest problems)
$O(n \log n)$ is expected to score about 60-70. (Use counting sort for small alphabet size)
$O(n)$ without tweaks is expected to score about 80-90.
$O(n)$ with tweaks is expected to score 100. (This is meant for fun only :)

Gambar A.4 Permasalahan Suffix Array pada Situs SPOJ

(Halaman ini sengaja dikosongkan)

BIODATA

Tracy Filbert Ridwan, lahir di Malang pada tanggal 4 Februari 1993, anak pertama dari 3 bersaudara. Penulis telah menempuh pendidikan formal mulai dari TK Santa maria, SD BPK Penabur Cirebon, SMP 1 BPK Penabur Cirebon, SMA 1 BPK Penabur Cirebon, dan terakhir sebagai mahasiswa Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember. Penulis mungkin bertampang sangat serius tapi jangan segan untuk menyapa apabila bertemu dengan penulis di suatu tempat karena sebenarnya penulis adalah seseorang yang sangat ramah.

Penulis sempat membantu mengurusi jalannya OSN Komputer 2012/13, Pelatnas II TOKI 2014/15 di ITS dan juga Pelatnas III TOKI 2015 di IPB. Selain itu penulis juga merupakan *boikoters* HMTC, asisten Pemrograman Terstruktur (2012-2014), asisten Algoritma & Struktur Data (2013-2014), mantan pemenang medali perunggu GeMasTIK 2013 Sub-Div Debugging, dan mantan pemenang medali perak GeMasTIK 2014 Sub-Div Pemrograman.